INRIA

UNITE DE RECHERCHE
INRIA RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 959

Programme 2

TROIS IMPLANTATIONS DU RECUPERATEUR DE MEMOIRE DE LA MACHINE MALI

Michel LE HENAFF
Hervé SANSON

Décembre 1988

TROIS IMPLANTATIONS DU RECUPERATEUR DE MEMOIRE DE LA MACHINE MALI

Michel LE HENAFF - Hervé SANSON

Publication Interne n° 443

Décembre 1988

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n° 443 - Décembre 1988 - 118 Pages

TROIS IMPLANTATIONS DU RECUPERATEUR DE MEMOIRE DE LA MACHINE MALI

Michel LE HENAFF - Hervé SANSON

RESUME

Le présent document est la réunion de deux rapports de *projets de DEA* effectués durant le printemps et l'été 1988 dans l'équipe MALI au centre INRIA de Rennes.

Trois versions logicielles, de la mémoire MALI, correspondant à trois schéma différents de récupération de mémoire, y sont présentées. Des mesures sont effectuées à propos d'une utilisation de ces versions de MALI par un même interpréteur Prolog, conçut à l'aide de MALI indépendamment de tout mécanisme de récupération de mémoire, compatible avec PrologII du GIA Marseille, et que nous appelons PrologII/MALI. Les résultats sont comparés avec une version de MALI existant antérieurement mettant en œuvre un schéma de récupération "à saturation" utilisant un algorithme "à la Baker".

Le premier rapport décrit la mise en œuvre de deux nouveaux schéma de récupération "à saturation", l'un utilisant un algorithme "à la Morris", l'autre un algorithme "à la Lisp2". Le deuxième rapport présente la mise en œuvre d'un schéma de récupération pseudo-parallèle "incrémental" utilisant un algorithme "à la Baker" qui est, grâce à ses deux demi-espaces, le seul à permettre le parallélisme ou le pseudo-parallélisme entre interprétation et récupération.

THREE IMPLEMENTATION OF THE GARBAGE COLLECTOR OF THE ABSTRACT MEMORY MALI

ABSTRACT

This paper is composed of two reports about two different *DEA projects* which took place at INRIA research center of Rennes, in MALI team, during spring and summer 1988.

Three software versions, of the memory MALI, corresponding to three different garbage collection schemes, are presented. Some measurements are made which concern the use of these MALI versions by one single Prolog interpreter, compatible with PrologII from GIA Marseille, which we call PrologII/MALI and which has been designed with MALI independantly from any garbage collection mechanism. The results are compared with a previous version of MALI which implements an "on memory overflow" garbage collection scheme which uses an "à la Baker" algorithm.

The first report describes the implementation of two new "on memory overflow" schemes, one using an "à la Morris" algorithm, the other using an "à la Lisp2" algorithm. The second report presents an implementation of a pseudo-parallel "incremental" scheme using an "à la Baker" algorithm which, thanks to its two semispaces, is the only one to allow some parallelism or pseudo-parallelism between interpretation and garbage collection.

Partie I

Mise en oeuvre d'un algorithme Baker incrémental

dans le système MALI.

Hervé Sanson

Je voudrais exprimer ma reconnaissance à Yves Bekkers qui m'a accueilli dans l'equipe MALI et a bien voulu me guider dans la réalisation de mon travail.

Je tiens à remercier Serge Le Huitouze qui n'a jamais hésité à sacrifier son temps pour me prodiguer une aide précieuse et des conseils toujours bienvenus.

Je remercie également Olivier Ridoux pour sa constante disponibilité et pour ses explications judicieuses.

Je voudrais aussi remercier Lucien Ungaro qui m'a apporté ses lumières bénéfiques sur les mystères de la version matérielle.

Merci également à Dominique Py qui a accepté de participer au jury de soutenance de ce stage.

Enfin, mes remerciements vont bien sûr à tous les autres membres de l'équipe MALI, dans laquelle j'ai eu le plaisir de faire ce stage de D.E.A.

Table des matières.

Introduction.

La machine MALI (Machine Adaptée au Langages Indéterministes) est un système destiné à servir à la mise en oeuvre des langages de programmation logique. Elle offre un certain nombre de commandes et est munie d'un recupérateur de mémoire autonome. Les versions logicielles de MALI, n'étaient dotées jusqu'alors que de récupérateurs à saturation.

On présente dans ce rapport l'implantation dans MALI d'un récupérateur utilisant l'algorithme de Baker et fonctionnant de manière incrémentale.

Cette implantation soulève des problèmes liés au parallélisme et nous expliquons la façon dont nous les avons résolus.

Enfin, nous mesurons les performances de ce récupérateur en fonction de nouveaux paramètres liés à l'introduction de la méthode incrémentale.

Chapitre 1

Généralités sur MALI.

MALI [BEK 86] (Machine Adaptée aux Langages Indéterministes) est une machine abstraite dont la sémantique est appropriée à la mise en œuvre des langages de programmation logique. Cette machine utilise une logique d'utilité qui a été prouvée spécifique aux langages sans affectation pratiquant une recherche en profondeur d'abord [RID 86].

MALI existe actuellement sous deux formes, matérielle (bi-processeur) et logicielle (ecrite en C). De plus, un interpréteur Prolog a été implémenté avec MALI comme support.

Les caractéristiques principales de MALI, sont une politique d'utilisation de la mémoire et une gestion automatique et optimale de cette mémoire par un récupérateur autonome.

On peut donc discerner dans MALI deux types d'activités :

- Le service : Tout ce qui a trait à la recherche en profondeur : installation et suppression de points de reprise, création et substitution de variables, retour-arrière. De plus la possibilité de construire ou parcourir des termes représentant l'état de l'interprétation.

- La récupération : Procédure qui étant donnée la sémantique du langage, découvre les structures de données devenues inutiles à l'état d'interprétation, libère la mémoire qu'elles occupent, et conserve les termes considérés comme utiles.

Les objets offerts par MALI.

MALI offre un certain nombre d'objets à l'utilisateur. Ces objets sont :

L'Atome,
Le Construit,
Le Nuplet,
La Variable,
La Variable à attribut,
Le Niveau.

Ces objets sont représentés dans MALI par des cellules, structures possédant deux champs, un champ *indicateur* et un champ *info.*

Le champ *indicateur* permet de connaître le type de l'objet.

Dans le cas d'un atome, le champ *info* représente la valeur de cet atome, et pour les autres objets, il correspond à un accès vers les différents constituants de l'objet.

Opérations de parcours et création de termes.

MALI permet à l'utilisateur de construire des termes, de les parcourir, de les dupliquer, ou encore de les comparer.

La gestion des piles nécessaires à ces commandes est interne à MALI. En aucun cas, l'utilisateur n'a à se préoccuper de gérer ces piles. Cette gestion fera d'ailleurs l'objet d'une étude plus poussée dans un chapitre ultérieur, car son implémentation est dépendante du système de récupération.

La logique d'utilité de MALI.

La logique d'utilité correspond à une formule qui spécifie ce qui est utile. Prenons l'exemple de Prolog, on peut dire que l'état d'interprétation est un ensemble de résolvantes ; quelque soit la manière dont ces résolvantes sont représentées, ce qui est utile correspond à ce qui contribue à représenter ces résolvantes.

On s'aperçoit de deux choses :

- Certaines variables peuvent ne plus avoir d'occurence dans aucune résolvante,

parce qu'elles sont substituées dans tous les états de liaison (on ne les "voit" donc plus). Dans ce cas, on remplace physiquement ces variables par leur valeur de liaison. Ceci constitue la dévariabilisation (ou constantisation).

- D'autres variables peuvent être substituées dans un certain état de liaison mais elles ne sont jamais accédées dans cette état. Leur valeur de liaison n'a alors plus aucun interêt et donc on décide de les remettre dans leur état libre. On appelle ceci la libération de variable (on peut dire que cette libération est anticipée dans la mesure où elle intervient avant la reprise du niveau dans lequel elle apparaît libre).

Une stratégie de recherche en profondeur d'abord (comme c'est le cas pour Prolog) permet d'avoir un ordre sur les résolvantes, lié d'ailleurs à un ordre sur les états de liaison. En particulier un état plus récent contient plus de liaison qu'un autre plus ancien.

L'idée est que, dans le cas d'une recherche en profondeur d'abord, il existe un algorithme mettant en œuvre la dévariabilisation ainsi que la libération, qui ne parcourt qu'une seule fois les termes partagés.

La gestion de la pile de recherche.

C'est MALI qui assure le contrôle de la recherche en profondeur d'abord (installation ou suppression de points de reprise, création et substitution de variables, et retour-arrière). En fait, MALI gère tout ce qui a trait au non-déterminisme.

La pile de recherche est composée de *niveaux* (cf. structures de données manipulées par MALI) contenant en particulier des termes sauvegardés par l'utilisateur. Ces termes sont sauvegardés dans leur représentation courante (on appelle ceci le partage "OU"). Quand un niveau est empilé dans la pile de recherche, il constitue un point de reprise, s'il est depilé on parle alors de retour-arrière et, dans le cas de Prolog, une coupure ("cut") correspond à supprimer des éléments de la pile de recherche.

Afin de garder le souvenir des modifications apportées aux représentations, on utilise une *traînée* : celle-ci correspond à la liste des modifications subies par les variables entre deux niveaux de sauvegarde.

Ainsi, le tronçon de traînée entre deux niveaux est formé du chaînage des variables ou variables à attribut qui auront été liées entre la sauvegarde du premier niveau et celle du deuxième, et des éventuels niveaux sauvegardés après le premier, mais coupés par la suite.

Le contrôle de la récupération mémoire.

La machine MALI est concue de façon à ce qu'un recupérateur de mémoire soit facilement réalisable. Cette récupération est considérée comme une procédure appelée par l'utilisateur et dont les paramètres sont les noms des termes utiles. En l'occurence cet appel est nommé *réduire*. Il constitue en fait une synchronisation entre l'interprétation (et donc la consommation) et la récupération.

Mais un appel à réduire n'implique pas forcément le déclenchement de la récupération. Suivant la mise en œuvre du récupérateur l'algorithme du réduire n'est pas le même. En particulier on a un comportement différent suivant que le récupérateur est séquentiel ou qu'il fonctionne de manière parallèle (un chapitre étudie d'ailleurs précisément les différentes possibilités lors d'un appel à réduire pour un système de récupération utilisant la technique de Baker Incrémental).

Les appels à réduire doivent se faire quand même relativement fréquemment ; dans le cas de Prolog la période la plus propice est le pas de résolution.

Pour ce qui est de la récupération mémoire proprement dite, on s'aperçoit que le rôle du récupérateur consiste en la détection des termes, des niveaux de sauvegarde, et des éléments de traînée devenus inutiles.

Ceci peut arriver dans les cas suivants :

- perte d'accès lors d'un *réduire* : Les termes qui ne sont pas des sous-termes des paramètres du réduire sont inutiles s'il ne servent pas pour un terme sauvegardé.
- perte d'accès suite à une modification de la pile de recherche lors d'une coupure ou d'un retour arrière.
- Libération anticipée ou dévariabilisation telles qu'on les a vu précédemment suivant la logique d'utilité de MALI.

En ce qui concerne les pertes d'accès, la récupération doit en particulier effectuer un parcours des germes de récupération (paramètre du réduire) et des termes sauvegardés. Ceci se décompose en deux phases : une phase de "marquage "qui détecte les représentations utiles, puis une phase de "ramassage" pour collecter celles qui restent et qui sont donc inutiles.

De plus le récupérateur doit examiner l'état des variables en fonction de la traînée et de la pile de recherche pour appliquer la logique d'utilité.

Le parcours se faisant des résolvantes les plus récentes vers les plus anciennes, on

detecte la libération et la dévariabilisation de la manière suivante :

- Pour la libération : quand on explore la traînée, si on rencontre une variable qui n'a pas été marquée depuis des niveaux plus récents, on peut la remettre dans son état libre.

- Pour la dévariabilisation : Toujours lors de la visite de la traînée, si la variable concernée a été liée dans le même niveau que celui où elle est née, on peut la "constantiser", c'est-à-dire la remplacer par sa valeur de liaison. La présence d'une information sur le niveau de naissance de la variable est donc nécessaire.

Chapitre 2

Principe de l'algorithme de récupération de Baker.

Le récupérateur de mémoire.

La tâche d'un récupérateur consiste à rassembler les portions de mémoire inutiles.

Ce processus travaille généralement en deux phases, l'une consistant à détecter les portions de mémoire utiles et qu'il faut donc préserver, et l'autre consistant à rassembler les portions inutiles et à les rendre accessibles à l'utilisateur.

Ce sont les méthodes employées pour réaliser ces deux phases qui différencient les systèmes de récupération.

Une des particularités de la méthode de récupération que nous allons présenter repose sur le fait qu'elle rassemble ces deux phases en une seule.

De plus, elle impose une certaine topologie de la mémoire. Grâce à cette topologie particulière, il est possible de concevoir un algorithme en parallèle.

Le principe de l'algorithme.

La méthode de récupération de Baker [BAK 78] est basée sur une vision de la mémoire différente des autres récupérateurs. En effet, elle impose comme condition de départ que l'espace mémoire soit partagé en deux espaces de tailles égales.

Durant l'exécution du programme utilisateur, tous les éléments sont localisés dans l'un des deux demi-espaces. Quand la récupération est sollicitée, on passe en paramètre

un certain nombre de termes utiles qui sont les racines d'accès.

De plus, le demi-espace contenant les éléments est appelé "ancien espace" ("fromspace") et l'autre demi-espace, "nouvel espace" ("tospace").

Les termes considérés comme utiles sont alors parcourus et au lieu d'être tout simplement marqués, ils sont recopiés dans le nouvel espace. Une "nouvelle adresse" est laissée dans l'ancien exemplaire, et si à un quelconque moment, on reconnaît un élément qui pointe sur un ancien exemplaire, on se sert de cette "nouvelle adresse" pour mettre l'élément à jour.

La fin de la phase de recopie est détectée quand toutes les cellules accessibles à partir des termes utiles du début ont été envoyées dans le nouvel espace et que toutes les références ont été mises à jour. On verra ultérieurement comment on détecte véritablement la terminaison.

Une fois que le nouvel espace a recueilli tous les éléments accessibles et que l'ancien espace ne contient donc plus que des termes utiles copiés et des termes inutiles, le ramassage est considéré comme fait. Il ne reste plus qu'à commuter le rôle des deux espaces, l'ancien devenant le nouveau, et vice-versa.

On voit, dès lors qu'on peut considérer la phase de ramassage comme instantanée (ce qui revient à considérer la vitesse de marquage comme infinie).

Cette méthode est donc simple et élégante puisque :

- Elle ne requiert qu'une seule phase effectuant le parcours et la copie des termes utiles au lieu d'une phase de marquage véritable et deux ou trois parcours linéaires de la mémoire correspondant au ramassage et au tassage pour les autres systèmes de récupération.

- Elle se passe de pile de parcours, les éléments déjà recopiés servant eux-mêmes de pile, grâce aux accès qu'ils ont sur les autres termes.

On peut donc considérer que la procédure de récupération est de complexité temporelle linéaire avec le volume de mémoire utile, puisque seuls les termes accessibles à partir des racines donnés au début, donc en fait tous les termes utiles, seront recopiés.

L'introduction de deux pointeurs.

La pile est evitée grâce à l'utilisation de deux pointeurs qu'on appellera le *curseur Baker* et le *curseur de copie.*

Le curseur de copie est incrémenté par le transfert des éléments d'un espace à l'autre. Il repère l'endroit du nouvel espace où on devra copier le prochain élément.

Le curseur Baker recherche les éléments dans le nouvel espace qui ont été copiés et les met à jour en provoquant la copie des objets auxquels ils font référence dans l'ancien espace si ceux-ci n'ont pas été copiés, ou en se contentant de mettre à jour cette référence si l'élément sur lequel elle pointe a déja été recopié dans le nouvel espace.

Le curseur Baker est initialisé pour pointer sur le début du nouvel espace à chaque inversion des deux espaces et il est incrémenté quand il a mis à jour l'élément sur lequel il pointe.

A n'importe quel moment, donc, les cellules entre le curseur Baker et le curseur de copie ont été transférées d'un espace à l'autre, mais les éléments qu'ils référencent n'ont pas été remis à jour.

Ainsi, quand le curseur Baker est égal au curseur de copie, on sait que tous les éléments accessibles ont été recopiés dans le nouvel espace et que toutes leurs références sont dans le nouvel espace.

Conclusion.

En dehors des avantages, vus précédemment, qui sont la linéarité par rapport aux termes utiles, et l'absence d'une gestion de pile de parcours, un des autres avantages de ce système de récupération est qu'il ne contraint en aucun cas la taille des éléments à être fixe. Au contraire, il considère la mémoire comme une suite d'emplacements sans aucune distinction et ne fait pas à priori d'hypothèses sur le type de l'élément sur lequel il pointe.

Cependant, tout système a ses inconvénients. Pour la méthode de Baker, ils sont les suivants :

- La condition de départ qui est de diviser la mémoire en deux est bien sûr restrictive quant à la taille de la mémoire disponible pour l'allocation, avant le déclenchement d'une

récupération.

- Le fait de recopier les éléments en fonction des accès dont ils font l'objet bouleverse totalement l'ordre de création de ces objets.

En effet, la zone entre le curseur Baker et le curseur de copie est gérée comme une file et on a donc un parcours en largeur d'abord.

Malgré cela, un des gros avantages de la récupération à la Baker, est d'offrir la possibilité d'une implantation en temps réel. En effet, l'idée est qu'on peut considérer la visite d'un élément et sa recopie éventuelle comme une opération atomique et indépendante. Le parallèlisme est donc possible et c'est la mise en place d'un système de récupération en temps réel dans la machine MALI qui sera l'objet de notre étude.

Chapitre 3

L'implantation du système de récupération de Baker Incrémental dans MALI.

Introduction.

Le problème consiste donc à implanter dans MALI un récupérateur utilisant la technique de Baker et fonctionnant de manière incrémentale.

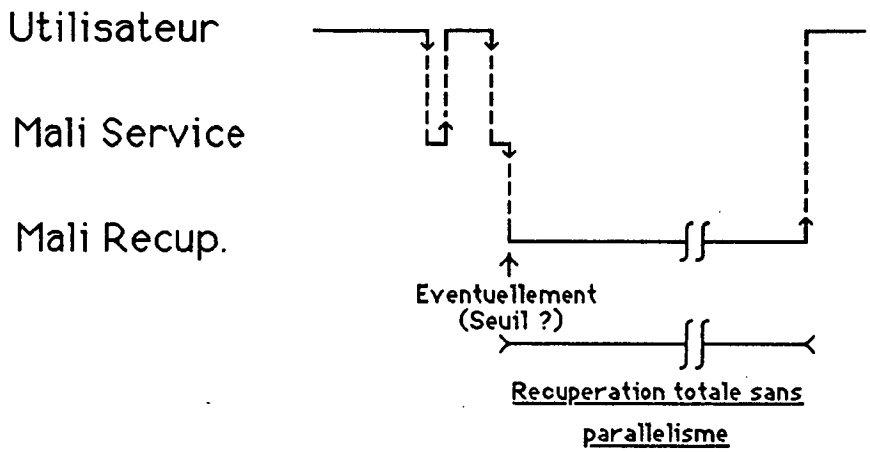
On peut considérer que la motivation tient sa source dans l'historique de MALI. En effet, MALI était conçue au départ pour fonctionner sous une forme matérielle qui utilise deux processeurs, l'un s'occupant de l'interprétation, l'autre de la récupération. Pour des raisons qu'on a pu voir précédemment, l'algorithme de Baker a donc été choisi. Puis l'idée est venue de donner à Mali, une forme logicielle écrite en C, dans laquelle on a implanté une version à saturation de l'algorithme de Baker. Il était donc logique d'utiliser à nouveau ce système pour implanter une version parallèle en monoprocesseur dans MALI.

Cette implantation est non triviale dans la mesure où l'introduction du parallélisme fait intervenir de nombreux changements autant sur la récupération que sur la gestion des piles dynamiques de MALI.

Avant de s'intéresser à la version incrémentale, on décrit rapidement les autres modes de garbage (saturation, bi-processeur) :

GARBAGE A SATURATION

(Version Logicielle)



Ce système existe dans une version logicielle de MALI.

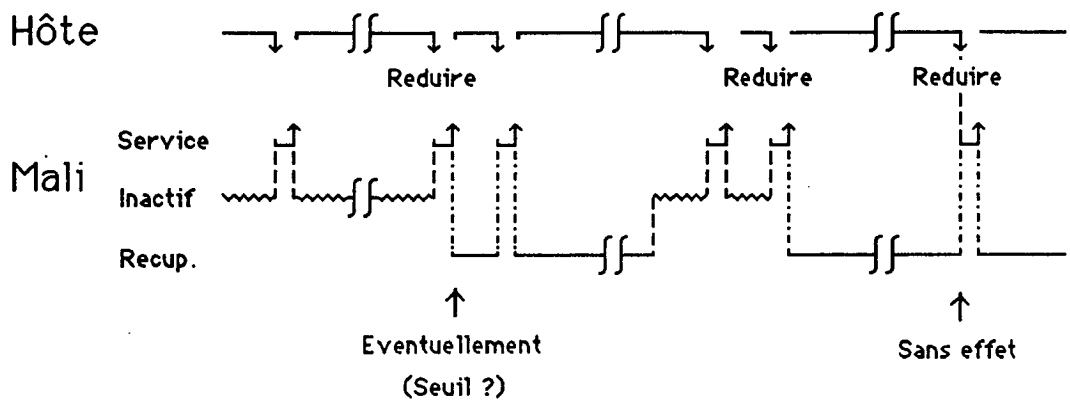
L'utilisateur fait appel au service tant qu'il le peut. Au bout d'un certain temps, l'allocation dépasse un certain seuil, et alors, on déclenche le récupérateur qui fait son travail en exclusion totale.

Lorsqu'il a fini de récupérer, la ressource est rendue à l'utilisateur jusqu'à la récupération suivante, etc...

En clair, la ressource n'étant pas partagée, l'utilisateur ne peut pas fonctionner en même temps que le récupérateur. Il reste bloqué jusqu'à la fin de la récupération.

BIPROCESSUS

(Version Hard)

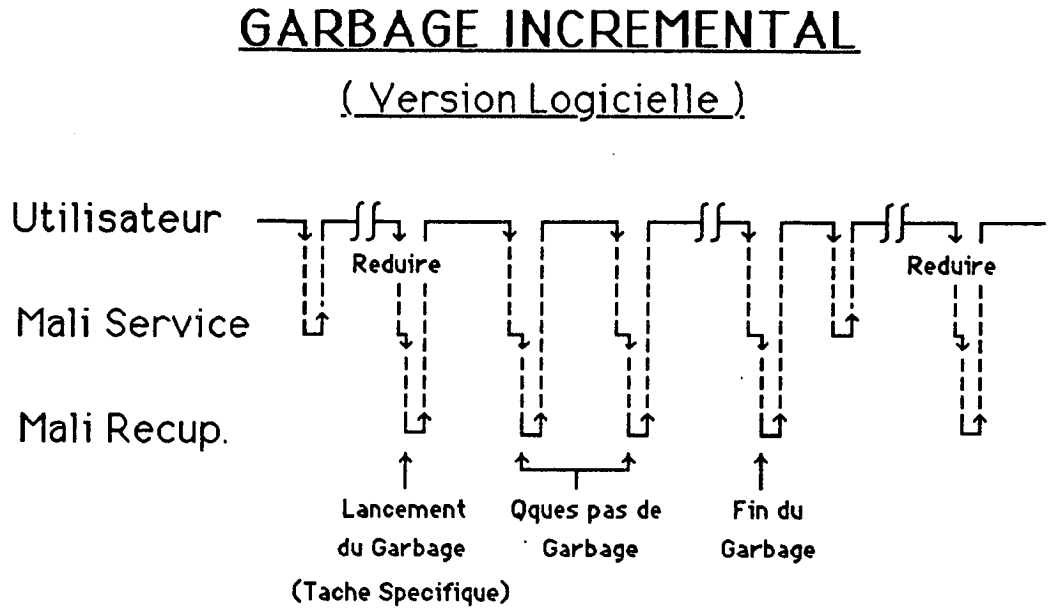


Ce mode de garbage utilise totalement le temps-réel. Le service et la récupération sont deux processus différents et ne possèdent pas de ressource commune.

Quand, à l'appel d'un réduire, on décide de déclencher la récupération, le processus récupérateur se met en marche. Il effectue son travail sans contrainte de la part du service et signale seulement sa terminaison. Tous les autres appels à réduire arrivant pendant la récupération, sont ignorés.

Le problème qui se pose dans cette version, est de régler la vitesse de récupération par rapport à celle de la consommation et de déterminer un seuil de déclenchement.

La recupération incrémentale : un pseudo-parallélisme.



La spécificité de la version incrémentale tient dans le partage de la ressource UC. La récupération et le service sont alors assimilés à deux coroutines.

Le problème se pose alors de savoir à quel moment la ressource sera donnée au récupérateur. Deux solutions sont possibles : Le récupérateur avance d'un certain nombre de pas de garbage, soit lors de l'appel à réduire, soit en liaison avec l'allocation.

C'est la deuxième solution que nous avons choisi. On comprend alors tout à fait le terme "incrémental" : le récupérateur "incrémente" son travail à chaque allocation.

Le système incrémental, comme tout système ayant trait au parallélisme, impose, de plus, un grain d'exclusion, en l'occurence une opération atomique.

Dans notre cas, on considère que cette opération est correspond à la visite d'une cellule par le curseur Baker. Cette visite sera entièrement décrite par la suite.

Cette visite constitue donc ce qu'on appelle un pas de garbage. Le nombre de pas de garbage est déterminé en fonction d'un rapport consommation / récupération.

Ceci permet d'une certaine certaine manière de régler la synchronisation entre le processus de consommation et celui de récupération.

Le rapport consommation / récupération fonctionne de la manière suivante : on a un nombre *c* ayant trait à la consommation, et un nombre *r* lié à la récupération. Quand on aura alloué *c* cellules, alors on pourra faire *r* pas de garbage.

Avant de s'interesser à la récupération proprement dite, on donne un aperçu de la représentation des objets offerts à l'utilisateur par MALI dans le cas du système de récupération de Baker incrémental.

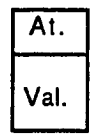
La représentation des informations dans MALI.

On a pu voir que les objets offerts à l'utilisateur étaient représentés par des *cellules* (cf. chapitre 1), structures contenant un champ *indicateur* et un champ *info*.

On rappelle que l'indicateur donne la nature de l'objet, et que le champ info représente, dans le cas d'un atome, sa valeur, et pour les autres objets, une référence sur ses constituants.

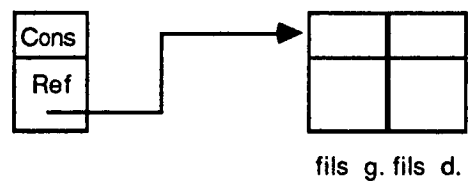
On présente ici un aspect plus détaillé de ces objets.

L'atome :



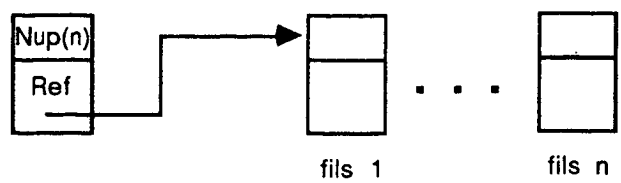
C'est le seul objet dont le champ info contient directement la valeur et non une référence.

Le construit :



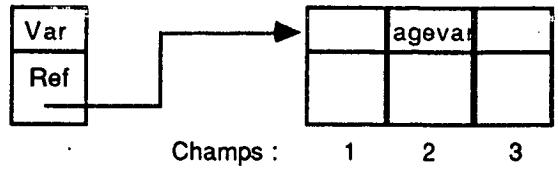
La référence pointe sur le premier terme, en l'occurence le fils gauche du construit.

Le nuplet :



La taille du nuplet figure dans le champ indicateur et la référence est sur le premier fils.

La Var :

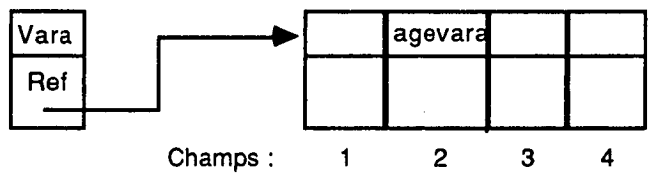


- Le champ 1 est le champ valeur de liaison. Il contient le nom de l'objet avec lequel elle est liée. Si elle n'est pas liée son champ indicateur contient le littéral *libre.*

- Le champ 2 est le champ niveau de naissance. Son champ indicateur contient le tag *agevar* et son champ info contient le nom du niveau sommet à sa création.

- Le champ 3 est le champ traînée. Il contient le nom de l'objet avec lequel la variable est chainée dans la traînée si elle est dans un état lié.

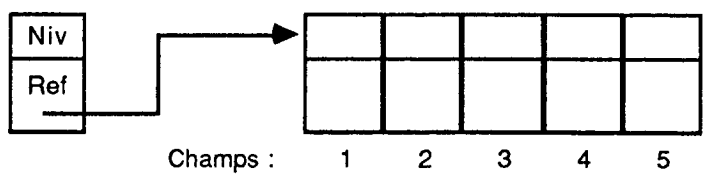
La Vara :



Les trois premiers champs sont les mêmes que ceux de la var : valeur de liaison, niveau de naissance, traînée.

Le champ 4 est le champ attribut. Il contient le nom de l'attribut de la variable. De par la logique d'utilité de MALI, ce champ n'a une signification que si la variable est dans son état libre.

Le niveau :



Le champ 1 est le champ indication d'activité. Son champ indicateur contient soit le littéral *coupé,* soit le littéral *actif.* Son champ info, quant à lui, contient la référence du niveau avec lequel celui-ci est chaîné.

Le champ 2 est le champ traînée. Il a le même rôle que le champ traînée d'une variable.

Les champs 3, 4 et 5 sont les germes de sauvegarde de l'utilisateur. Ils contiennent des noms d'objets donnés par l'utilisateur.

La récupération de mémoire.

On a vu précédemment que le grain élémentaire d'avancement du récupérateur était constitué par la visite d'une cellule par le curseur Baker.

De même, on a vu qu'il y avait une gestion des processus de récupération et de consommation dans la mesure où la visite par le curseur Baker se fait en liaison avec l'allocation de nouvelles cellules. En particulier, on décide du nombre de cellules qu'on doit récupérer en fonction du nombre de cellules qu'on a alloué. Ceci sera d'ailleurs l'objet d'une étude sur les mesures de performances.

Le système de récupération de Baker va s'appliquer, dans MALI, d'une manière un peu particulière. En effet, le parcours et la recopie des termes utiles va se faire en fait pour chaque niveau de sauvegarde, les germes de sauvegarde de ces niveaux servant de termes initiateurs au parcours d'un niveau.

On détecte donc la fin du parcours d'un niveau, quand le curseur de copie est égal au curseur Baker. Mais ceci ne prouve rien quant à la terminaison de la récupération. Celle-ci est détectée quand le niveau à parcourir est égal au niveau "plancher", c'est à dire le niveau le plus vieux.

Par contre, une fois qu'un niveau a été entièrement recopié, on doit effectuer quelque chose qui n'a rien à voir avec le système de Baker, c'est-à-dire appliquer la logique d'utilité de MALI. Ceci se fait en explorant la traînée.

On considère que l'opération qui consiste à visiter un maillon de traînée est atomique, de la même façon que la visite d'une cellule par le curseur Baker.

Le curseur de copie, quant à lui, se comporte de manière quelque peu différente de ce qu'on a vu précédemment, dans la mesure où il ne dépend plus uniquement de la visite des objets par le curseur Baker. Il a une autre manière d'avancer liée à un phénomène que nous allons voir maintenant.

En effet, la notion de parallélisme implique qu'à un moment donné de la récupération, il y a des termes utiles aussi bien dans l'ancien espace que dans le nouveau. On voit qu'il est nécessaire que MALI ne connaisse que des références dans le nouvel espace.

On en déduit l'invariant suivant : "Toute commande de MALI doit ramener une référence dans le nouvel espace".

On introduit donc la notion pour un un objet d'être "à jour" ou non.

Ainsi pour toute commande de MALI, si cette commande doit faire un accès sur un objet qui se trouve dans l'ancien espace, on doit visiter cet objet de la même façon qu'on le fait quand le curseur Baker pointe sur un objet dont les constituants sont dans l'ancien espace. On ramène ensuite la nouvelle référence sur cet objet.

L'algorithme de la visite est le même, dans le cas où l'objet à visiter est l'objet sur lequel pointe le curseur Baker, que dans le cas où celui-ci n'est autre que l'objet qu'on veut accéder.

Nous allons maintenant décrire précisément l'algorithme de cette visite.

Le parcours d'un niveau par le curseur Baker.

Au cours de son parcours des niveaux, le curseur Baker peut pointer sur différents noms d'objets de MALI : atomes, construits, nuplets, variables, variables à attribut, niveaux. Baker considère la mémoire comme une succession de cellules, dans la mesure où il ne sait pas, sauf dans certains cas, le type de l'objet dans lequel il se trouve. Sa progression, à part dans les cas d'exception, se fait donc cellule par cellule. L'opération de visite d'une cellule est considérée comme atomique.

On rappelle que les objets manipulés par MALI sont des cellules, c'est-à-dire une structure contenant un champ *indicateur* et un champ *info*.

Nous allons donc observer les différents travaux de recopie, suivant les noms d'objets sur lesquels pointe le curseur Baker :

Le curseur Baker cite un nom d'atome

Ce cas est particulièrement trivial puisque le champ info du nom d'atome ne contient pas une référence, mais sa propre valeur. Il n'y a donc ni mise à jour ni recopie. Le curseur Baker se contente d'avancer à la cellule suivante.

Le curseur Baker cite un nom de construit

On trouve dans le champ info du nom de construit la référence sur son sous-terme gauche, le sous-terme droit occupant la cellule suivante. On s'intéresse au cas où la référence est dans l'ancien espace, sinon la référence étant à jour, il n'y a juste qu'à faire progresser le curseur Baker d'une cellule.

Si le champ indicateur du sous-terme gauche contient le littéral *renvoiL,* c'est que ce construit a déjà été recopié, et dans ce cas, le champ info du sous-terme gauche contient la référence de la copie. Il suffit donc de modifier le champ info du nom de construit pour qu'il pointe sur cette copie.

Dans le cas contraire, on alloue en zone de copie deux cellules, on y copie les sous-termes gauche et droit et on modifie le champ info du nom de construit afin qu'il référence cette copie. De plus on remplace le champ indicateur du sous-terme gauche de l'ancien exemplaire par le littéral *renvoi* et on modifie son champ info de telle sorte qu'il pointe lui aussi sur sa copie.

On fait ensuite progresser le curseur Baker d'une cellule.

Le curseur Baker cite un nom de nuplet

Le principe est le même que pour le construit, celui-ci ayant en fait la même structure qu'un nuplet de taille 2.

Pour copier un nuplet, s'il doit effectivement y avoir copie, on a donc juste besoin, comme information supplémentaire, de la taille de celui-ci (c'est-à-dire le nombre de ses sous-termes), information qui figure dans le champ indicateur, ceci bien sûr pour réserver en zone de copie la quantité de cellules nécessaire et pour effectuer la copie des sous-termes.

Le curseur Baker cite un nom de var ou vara

On trouve dans le champ info du nom de la variable, l'adresse de son premier champ, le champ valeur de liaison. On s'intéresse toujours au cas où cette adresse est dans l'ancien espace.

Si le champ indicateur du champ valeur de liaison contient *renvoiL,* on sait que les constituants de la variable ont déjà été recopiés et dans ce cas on modifie le champ info du nom de la variable avec l'adresse du nouvel exemplaire.

Autrement, on sait que c'est la première fois que l'on rencontre cette variable.

Si le champ indicateur du champ niveau de naissance vaut *constL,* c'est que la variable a été précédemment reconnue comme dévariabilisable. On remplace alors le nom de variable par le nom d'objet qui figure dans son champ valeur de liaison. Les champs de la variable perdant donc alors toute signification et ne necessite en aucun cas une recopie. De plus dans ce cas on n'avance pas le curseur Baker.

Autrement, on doit allouer dans la zone de copie trois cellules pour une variable ou quatre pour une variable à attribut et y recopier ses champs. On avance ensuite le curseur Baker d'une cellule.

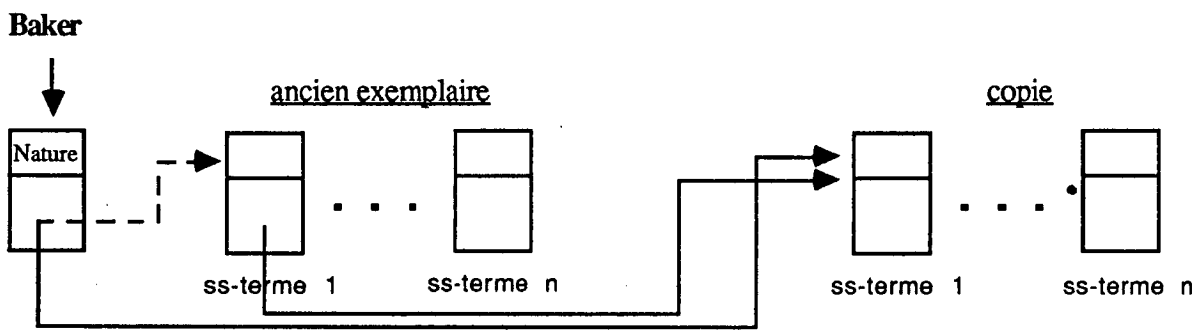
Le curseur Baker cite un nom de niveau

Dans le champ info du nom de niveau on trouve la référence au champ indication d'activité du niveau. Dans le cas où celui-ci n'est pas actif, on parcourt le chaînage des niveaux (on rappelle que la valeur de chaînage se trouve dans le champ info du champ indication d'activité) jusqu'à ce que l'on arrive sur un niveau actif. Ensuite on remplace la référence dans le champ info du nom de niveau par la référence sur le champ indication d'activité du premier niveau actif.

Si ce niveau est dans l'ancien espace, on alloue en zone de copie cinq cellules, et

on copie le niveau. On écrit le littéral renvoiL dans le champ indicateur du niveau recopié et on fait pointer son champ info sur le champ indication d'activité du nouvel exemplaire. De plus, on modifie le champ info du nom de niveau avec la référence sur le nouvel exemplaire.
Cette technique offre la certitude d'avoir recopié tous les niveaux actifs, à la fin du parcours d'un niveau. En effet tous les niveaux actifs sont chaînés entre eux, le niveau sommet est copié dans le nouvel espace dès le début de la fournée, et, on vient de le voir, le chaînage des niveaux est considéré comme un accès de termes.

On a pu voir que le travail de recopie reste quand même relativement semblable quelque soit l'objet à recopier. La figure suivante donne une idée de ce travail sur un objet quelconque :



Cependant il peut arriver que le curseur Baker ne cite pas un nom d'objet, mais qu'il cite un champ d'un objet dont le champ indicateur contient une valeur spéciale. A priori rien ne lui permet de savoir de quel objet il s'agit, sauf dans certains cas.

Ce sont ces cas que nous allons étudier maintenant, ainsi que le travail fait par Baker quand il s'y trouve confronté :

Le curseur Baker cite le champ indication d'activité d'un niveau

1er cas : L'indicateur de ce champ est le littéral *coupéL*.

On se contente dans ce cas d'ignorer tout simplement le niveau coupé. Pour ce faire, il suffit tout simplement que Baker ne visite pas les autre champs du niveau. En fait, on fait progresser le curseur Baker de cinq cellules.

2ème cas : L'indicateur de ce champ est le littéral *actifL*.

Ici, on vérifie si le niveau obtenu par chaînage avec celui-ci est déja renvoyé, auquel cas on met à jour le chaînage vers le niveau renvoyé. Sinon, si la référence de chaînage pointe quand même dans l'ancien espace, on fait une copie du niveau chaîné et on installe le renvoi. A la fin, on fait là aussi progresser le curseur Baker de cinq cellules.

Le curseur Baker cite le champ niveau de naissance d'une variable
Le champ indicateur de la cellule citée par le curseur Baker contient le littéral *agevarL*. On trouve dans le champ info le niveau qui était le niveau sommet lors de la création de la variable. Si celui-ci n'est plus actif on parcourt le chaînage jusqu'au premier actif et on fait pointer le champ info sur ce niveau. On verifie ensuite si ce niveau est dans l'ancien espace et dans ce cas on en fait une copie et on installe le renvoi dans le champ niveau de naissance de la variable visitée et de l'ancien vers le nouvel exemplaire.
On avance ensuite le curseur Baker de façon à ce qu'il ignore le champ trainée de la variable (il progresse donc de deux cellules).

Le curseur Baker cite le champ niveau de naissance d'une variable à attribut
Le champ indicateur de la cellule citée par le curseur Baker contient le littéral *agevaraL*. Le traitement est le même que pour une variable jusqu'à la copie du niveau. Ensuite, dans le cas d'un variable à attribut, on doit retourner à la cellule précédente (c'est-à-dire au champ valeur de liaison) pour s'assurer de l'état de liaison de la variable. Si le champ indicateur de celle-ci contient le littéral *libreL* ou les littéraux *duplicataL* ou *déja_vuL* (pour des raisons de temps-réel : en effet, la variable pouvait être en phase de duplication) on doit visiter l'attribut maintenant. On s'arrange donc pour que le curseur Baker pointe sur le champ attribut lors de sa prochaine visite. Pour cela il suffit de le faire progresser de deux cellules.
Dans l'autre cas, on sait que la variable est liée et que son attribut sera alors visité lors du parcours de la traînée. On fait en sorte que Baker ignore le champ attribut, en l'avançant de trois cellules.

Pour toutes les autres valeurs que peuvent prendre les champs indicateur des cellules pointées par le curseur Baker (*ctrlL, libreL, duplicataL, deja_vuL, assocL*), on ne fait rien d'autre que de le faire progresser d'une cellule.

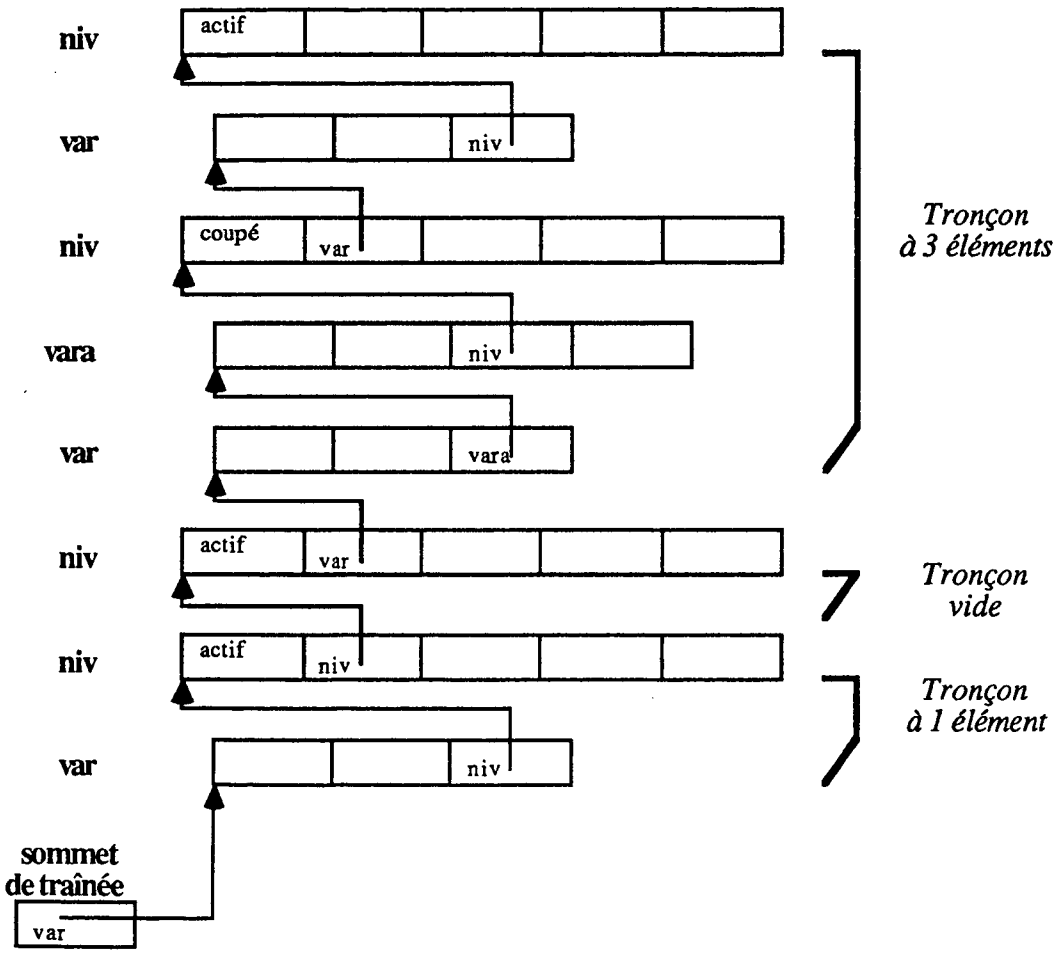
Le parcours de la traînée.

Rappel sur la trainée

La traînée est définie comme un chaînage entre des niveaux, des variables et des variables à attribut.

Seules les variables ou variables à attribut liées peuvent être présentes dans la traînée. C'est d'ailleurs le fait de les lier qui impose leur introduction dans la traînée.

On peut représenter la traînée de la façon suivante :



Le tronçon de traînée du niveau n est formé du chaînage des variables ou variables à attributs qui ont été liées entre la sauvegarde du niveau n et du niveau n+1, et des éventuels niveaux sauvegardés depuis le niveau n puis coupés avant la sauvegarde du niveau n+1. Le tronçon se termine par la référence au niveau n. Le sommet de traînée indique l'endroit où attacher le futur maillon de traînée.

L'algorithme de visite de la traînée

Le parcours d'un tronçon de l'ancienne traînée doit se faire quand Baker a fini la recopie des germes d'un niveau. Ceci survient quand on a l'égalité entre le curseur Baker et le curseur de copie.

Le traitement d'un maillon de traînée, tout comme la visite d'une cellule par Baker, est atomique.

On dispose à chaque fois de la référence sur le maillon de traînée à visiter et de l'endroit pionnier où accrocher le maillon après traitement si toutefois c'est utile.

On l'a vu précédemment, le maillon de traînée est soit un niveau, soit une variable liée, soit une variable à attribut liée. Nous allons donc observer le comportement du récupérateur dans chacun de ces cas.

Le maillon de traînée est une variable

Si le champ indicateur du champ valeur de liaison de la variable est le littéral *renvoiL*, c'est que la variable a déjà été copiée et dans ce cas, on sait qu'elle est est utile.

On teste alors si la variable est dévariabilisable. Si oui, alors sa présence dans la traînée n'est plus utile, et donc on se contente de positionner le prochain maillon à visiter sur l'objet avec lequel elle était chaînée (la référence de celui-ci se trouve dans le champ traînée de la variable).

Si la variable n'est pas dévariabilisable, il faut la chaîner comme nouveau maillon dans la nouvelle traînée et pour cela, on dispose de l'endroit pionnier qu'on met ensuite à jour avec le champ traînée de la variable.

Si la variable n'a pas été copiée, c'est qu'elle n'a pas été accédée depuis les niveaux déjà traités. Les niveaux inférieurs ne risquent plus de la voir liée, donc on la délie. Il suffit pour cela de mettre le littéral *libreL* dans le champ indicateur de son champ valeur de liaison.

Le maillon de traînée est une variable à attribut

Le traitement est le même que pour une variable sauf dans le cas où la variable à attribut est utile et non dévariabilisable. En effet, il faut alors visiter l'attribut, ceci n'ayant pas été fait lors de la rencontre de la variable liée, l'attribut n'ayant de sens que dans la situation libre de la variable.

Ensuite on chaîne celle-ci dans la nouvelle traînée, comme on le fait pour une variable ordinaire.

Le maillon de trainée est un niveau

Ce niveau est alors soit un niveau coupé, soit un niveau renvoyé (en effet, tous les niveaux actifs ont été copiés donc renvoyés, cf. parcours de la mémoire par Baker).

Dans le premier cas on l'ignore tout simplement, et on prépare la visite de son successeur dans la traînée.

Dans le deuxième cas, on doit vérifier si le renvoi est sur un niveau coupé ou actif.

Si c'est un niveau coupé, on sait que la coupure a eu lieu depuis le début de la fournée, et on procède alors comme dans le cas d'un niveau coupé et non copié.

Dans le cas d'un renvoi sur un niveau actif, on installe celui-ci dans la nouvelle traînée et on met à jour l'endroit pionnier. On vérifie ensuite si le niveau courant est égal au niveau plancher, auquel cas la fournée est terminée.

Dans le cas contraire on copie les germes de sauvegarde du niveau, ce qui provoque une différenciation entre le curseur Baker et le curseur de copie, et donc la reprise du parcours par le curseur Baker.

Le problème du recalage.

Il peut arriver, alors que le récupérateur marque un niveau, que l'utilisateur décide de modifier l'état de la pile de recherche, soit sous la forme d'une coupure, soit sous celle d'un retour-arrière.

Un problème, spécifique au parallélisme, peut alors se poser : le consommateur risque d'aller plus vite que le récupérateur. En effet, ce retour-arrière va faire que des objets que le recupérateur est en train de marquer, ainsi que des maillons de traînée, vont devenir alors inutiles.

En fait, on peut dire que la vision de la mémoire qu'avait le récupérateur au moment du déclenchement se trouve bouleversée par le consommateur.

C'est à ce moment qu'intervient le "recalage".

Dans le cas de la coupure, le problème se règle assez simplement : il faut recaler si le prochain niveau à parcourir devient coupé. On met alors à jour celui-ci avec le premier niveau actif rencontré lors du parcours de chaînage des niveaux.

Dans le cas d'une reprise, le problème est beaucoup plus complexe. On évitera de rentrer dans des considérations techniques qui manqueraient d'interêt pour le lecteur.

Ce qu'il faut savoir, c'est que le recalage est détecté si le prochain niveau à parcourir est égal à l'ancien niveau sommet (c'est-à-dire le niveau sommet avant le "reprendre").

Si c'est le cas, on fait en sorte que le niveau à parcourir devienne le nouveau niveau sommet.

La traînée "vue" par le récupérateur n'ayant plus de sens, le prochain élément de traînée à visiter doit être celui figurant dans le champ trainée du niveau repris.

L'influence de l'implantation de Baker Incrémental sur la gestion dynamique de piles de parcours et de trace.

On rappelle tout d'abord l'origine et l'utilité de ces piles :

La comparaison de termes rationnels nécessite l'utilisation d'une pile de parcours de bi-nœuds pour plonger dans les deux termes à comparer, et d'une pile de trace pour garder le souvenir d'associations, ceci dans le cas de termes rationnels infinis.

La duplication de termes rationnels utilise, elle, une pile de parcours de rationnels pour plonger dans le terme à copier et une pile de trace pour la même raison que la comparaison.

Le parcours de termes rationnels s'appuie lui aussi sur une pile dynamique pour explorer les nœuds du terme à parcourir.

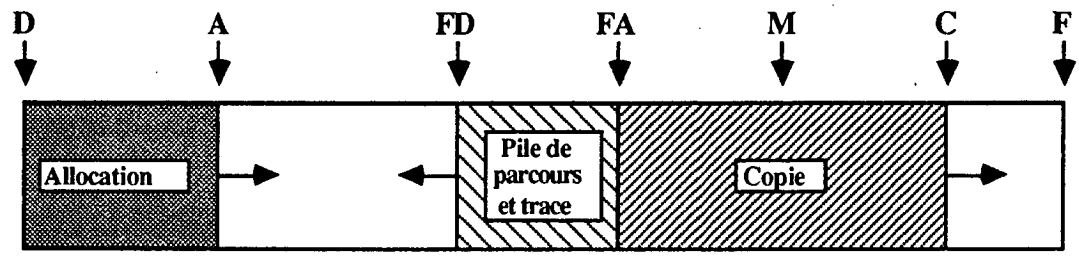
Un des domaines où l'implémentation de Baker Incrémental provoque des changements est justement la gestion de ces piles dynamiques de parcours et de trace. Dans la version Baker à saturation, on disposait de l'ancien espace pour la gestion de ces piles. En effet, on était sûr, la fournée étant effectuée en exclusion, que quand le service reprenait le contrôle, tous les termes utiles avaient été recopiés dans le nouvel espace ; on pouvait alors considérer l'ancien espace comme un espace "auxiliaire", utilisable pour des travaux tels la gestion de piles.

Par exemple, pour la duplication d'un terme rationnel on disposait d'une pile de parcours qui débutait à la fin de l'ancien espace et qui croissait vers le début, et d'une pile de trace qui commençait au début de l'ancien espace et qui allait vers la fin. Au moment où on déclenchait la récupération, on était sûr que les piles étaient vides, et le fait de travailler dans les deux espaces ne posait donc aucun problème.

La situation est bien différente dans le cas d'un sytème de récupération en temps-réel puisqu'il est difficilement possible d'utiliser l'ancien espace pour un quelconque travail, n'étant jamais certain de ne pas écraser des termes qui n'auraient pas encore été visités et recopiés dans le nouvel espace. On retrouve là une notion fondamentale du temps-réel qui est que, jusqu'à la fin d'une fournée, on utilise les deux espaces, ancien comme nouveau.

La solution choisie pour résoudre ce problème devait donc respecter l'obligation de gérer la pile dans le nouvel espace. Pour cela, on a décidé de coupler la pile de parcours et la pile de trace, tout en leur laissant un sommet respectif. A cela, il faut ajouter un invariant qui est que la pile de trace contient toujours plus d'éléments que la pile de parcours. Le front descendant de la pile commune à la trace et au parcours est donc représenté par le sommet de trace.

On peut représenter le nouvel espace de la façon suivante :



D : Pointeur sur le début de l'espace.

A : Pointeur de début d'allocation. Il représente la première place disponible pour l'allocation.

FD : Front descendant de pile. Il évolue suivant la pile de trace.

FA : Pointeur de fin de zone d'allocation (et donc de début de zone de copie).

M : Curseur Baker.

C : Curseur de copie. Il représente le premier objet disponible pour la copie.

F : Pointeur de fin d'espace.

Nous allons maintenant étudier un exemple de pile un peu plus dans le détail.

La pile associée à la duplication de termes rationnels.

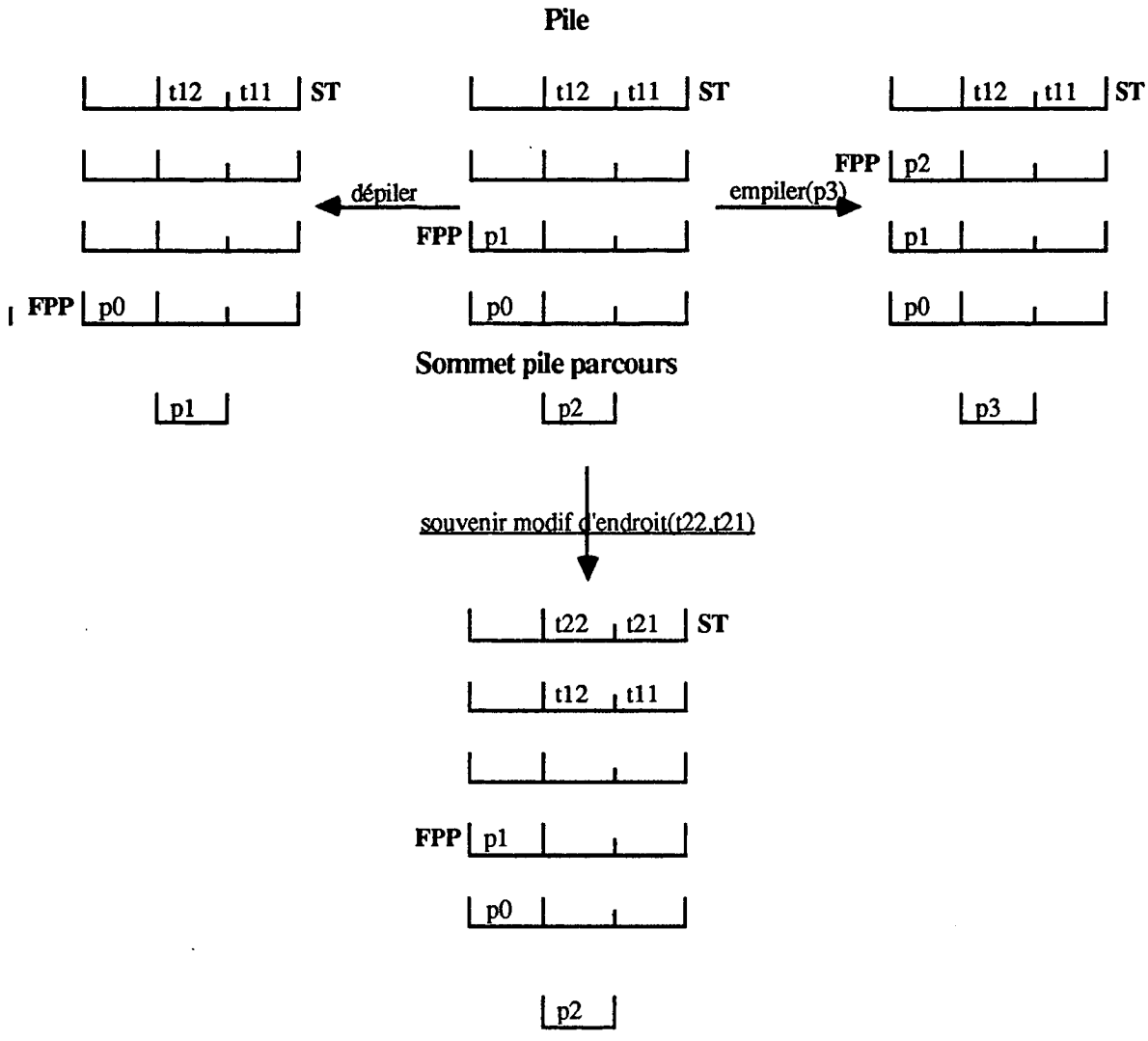
Chaque élément de cette pile est un groupement de trois cellules : une cellule pour le parcours, deux pour la trace.

Le sommet de pile de parcours est déporté dans une variable globale.

On a ci-après un aperçu de l'effet des diverses opérations sur la pile.

La seule opération à ne pas être détaillée est la restauration de trace, celle-ci ayant pour effet de dépiler toute la pile.

Les diverses opérations sur la pile dynamique de duplication de rationnels.



FPP : front de pile de parcours.

ST : sommet de trace.

Il est important de constater la totale indépendance entre la partie parcours et la partie trace, celles-ci ne dépendant que de leur sommets respectifs.

De même, il ne faut pas oublier, bien que la représentation soit trompeuse, que toute

progression de la pile va vers les adresses décroissantes.

Le sommet de trace, étant l'adresse la plus avançée dans la pile, est utilisé pour le test de collision éventuelle entre lui et le pointeur d'allocation. Initialement il précède le pointeur de fin d'allocation d'une cellule.

Il serait assez inutile de refaire ce schéma pour les autres piles, dans la mesure où la gestion reste sensiblement identique. Les seules différences sont les suivantes :

La pile associée à la comparaison de termes rationnels possède des éléments de quatre cellules, deux pour le parcours, et deux pour la trace.

La pile associée au parcours de termes rationnels possèdent des éléments d'une cellule. En effet, la notion de trace disparaît, et on a besoin d'une seule cellule pour le parcours.

Au niveau de l'état de mise à jour des éléments de la pile, pour éviter des problèmes dûs au temps réel, on a considéré l'empilement de valeurs comme un accès et on visite ainsi chaque valeur à empiler de façon à n'avoir dans la pile que des références dans le nouvel espace.

Les paramètres directement liés au déclenchement du récupérateur.

Trois paramètres permettent de régler le comportement du récupérateur. On distingue parmi ceux-ci, deux paramètres de fonctionnement, (1) la consommation maximale de mémoire entre deux *réduire*, c'est-à-dire entre deux inférences successives, (2) un seuil de déclenchement calculé en fonction d'un certain pourcentage d'occupation de la mémoire. Le troisième paramètre ne concerne pas MALI, il spécifie un seuil de saturation avant la fin (3), qui permettra à l'utilisateur d'accélérer la "mort" de la machine plutôt que de se laisser entraîner dans un fonctionnement dégradé.

On ne précisera pas ce qui se passe lors d'un déclenchement, une étude particulière lui étant consacrée.

(1) Le premier paramètre donne la consommation maximale entre deux *réduire* , il sert à spécifier un critère de bon fonctionnement de MALI. Il correspond à un nombre de cellules considéré comme un majorant du besoin en mémoire pour une étape, étape qui dans le cas de Prolog représente un pas de résolution.
Cette valeur intervient dans le calcul d'un seuil par rapport à la fin de la zone d'allocation. L'interprète se met en attente si, lors d'un appel à réduire, la taille restante de la zone d'allocation (portion comprise entre le pointeur d'allocation et celui de fin de zone) est inférieure à ce besoin pour une étape. Le parallèlisme entre le récupérateur et l'interpréteur est alors entravé, la totalité de la ressource UC étant consacrée à la récupération.
Le non respect de ce critère peut entraîner un message d'abandon de MALI. Notons alors que la situation n'était peut-être pas irrécupérable, mais ne sachant pas y remédier, on préfère provoquer tout de même un abandon.
A l'opposé, ce critère peut être violé sans entraver le bon fonctionnement de MALI (il suffit que la taille de la mémoire allouable au moment où le critère a été violé soit suffisante).

(2) Le deuxième critère de fonctionnement est le seuil de déclenchement. Ce seuil est spécifique au temps réel, et est confondu avec la notion de consommation maximale entre deux *réduire* pour les recupérateurs à saturation. Il correspond à un pourcentage d'occupation de la mémoire qui provoquera, ou non, le déclenchement du récupérateur.

Avec ce seuil, on calcule la taille minimale de la mémoire qui doit rester inoccupée en zone d'allocation pour éviter de déclencher le récupérateur. Il s'ensuit qu'on limite le nombre de déclenchements, et on assure par la même occasion un comportement plus "libéré" du service.
Ce seuil doit être choisi en fonction des vitesses du marqueur et du consommateur, et d'une borne supérieure de la quantité de mémoire à marquer. Il doit être tel que le marquage puisse se terminer avant que la mémoire disponible ne soit épuisée.
L'absence de ce seuil, bien que possible sans remettre en cause le fonctionnement de MALI, est difficilement acceptable, dans la mesure où ce fonctionnement se retrouve totalement dégradé, chaque *réduire* risquant de déclencher le récupérateur. Le nombre de fournées prend alors très vite des proportions énormes et le récupérateur monopolise largement le temps d'exécution, même lors de l'interprétation de programmes qui ne nécessitaient pas de récupération.

(3) Le dernier paramètre, enfin, n'est pas un critère de fonctionnement, mais plutôt un service offert à l'utilisateur pour ne pas se laisser entraîner dans un comportement vicié. C'est le seuil de saturation avant la fin, qui correspond lui aussi à un pourcentage de la mémoire.
On calcule l'espace des termes utiles à l'issue d'une fournée forcée lors d'un reduire. Si le volume des termes utiles, représenté en l'occurence par la taille de la zone de copie (en effet, la fournée ayant été exclusive, la zone d'allocation est vide), est supérieur au pourcentage donné par l'utilisateur,alors on considère qu'il est inutile de laisser se continuer une application pour laquelle on va passer beaucoup trop de temps en récupération. On décide alors de provoquer la "mort" de cette application plutôt que d'accepter un fonctionnement dégradé qui nuirait aux performances de l'interpréteur.

En résumé, il y a donc deux manières de saturer MALI :

1- La manière correcte, le dépassement du seuil de saturation avant la fin, qui se detecte lors d'un réduire après forçage du récupérateur.

2- La violation de la condition de consommation maximale entre deux réduire, qui se détecte en cas de manque d'espace lors d'une allocation.

Le déclenchement du récupérateur.

On va s'intéresser maintenant aux conditions de déclenchement du récupérateur : dans quel cas, et à quel moment décide-t-on de lancer une nouvelle fournée.

Le déclenchement, ne peut se produire que lors d'un *réduire*, instruction exécutée à chaque nouvelle inférence, c'est à dire à chaque fois que le moteur s'intéresse à la résolvante.

On va voir que lors d'un réduire on peut, ou déclencher le récupérateur, ou alors le forcer à terminer une fournée en cours, ou bien tout simplement ne rien faire.

Les paramètres liés au déclenchement et cités ici sont ceux détaillés précedemment.

Pour l'instant, intéressons nous aux éventualités qui font qu'on n'a pas déclenché de nouvelle fournée.

Il y a tout d'abord le simple fait qu'on ne déclenche jamais une fournée tant qu'une autre est en cours [5]. Pour envisager un déclenchement, il faut donc nécessairement que la fournée précédente soit terminée.

Dans le cas où il n'y a pas de fournée en cours, il faut tenir compte d'un des paramètres liés au déclenchement, à savoir le *seuil de déclenchement.* Ce seuil qu'on pourrait qualifier de "paresse" correspond à un pourcentage d'occupation de la mémoire, qui doit être franchi pour qu'un déclenchement s'effectue [6].

Si ce pourcentage est à zéro, le recupérateur ne connait aucune paresse, et se déclenche au premier réduire tel que la condition de terminaison d'une fournée soit vérifiée. Par opposition, le recupérateur se déclenche d'autant plus rarement que le seuil de paresse est plus grand, mais le travail de récupération doit être alors fait d'autant plus vite.

La variation de ce seuil, on le voit, pourra être prétexte à des mesures sur le temps d'exécution et sur le nombre de fournées déclenchées.

Voyons maintenant le cas ou une fournée est forcée à se terminer.

Un des autres paramètres liés au déclenchement est *la consommation maximale entre deux réduire* qu'on peut appeler aussi *besoin pour une etape.* Ce paramètre, représentant un majorant de la quantité de mémoire consommable entre deux inférences,

va influer sur le comportement du récupérateur de la manière suivante : si on s'aperçoit, à l'origine d'un *réduire*, que la partie restante en allocation est inférieure à ce seuil [1], il est impératif de bloquer l'interpréteur et de consacrer tout le temps à la récupération . A la fin de celle-ci, on amorce une nouvelle fournée, c'est-à-dire qu'on commute les deux espaces et qu'on determine la nouvelle zone d'allocation et la nouvelle zone de copie. On vérifie alors à nouveau que la nouvelle zone d'allocation dépasse le *besoin pour une étape* [2]. Si cette condition est toujours fausse, on finit de la même façon la fournée en bloquant l'interpréteur.

A l'issue de cette fournée, on se retrouve donc avec une zone d'allocation vide, puisque l'interpréteur n'a pas eu la possibilité de travailler. On s'assure alors que le volume des termes utiles correspondant ici à la taille de ce qui vient d'être copié ne dépasse pas un troisième paramètre, *le seuil de saturation* [3], et on amorce une nouvelle fournée. Dans le cas contraire, il ne reste plus qu'à provoquer l'abandon de l'interprétation [4].

Cette méthode permet de synchroniser le recupérateur et l'interpréteur dans le cas où on consomme plus vite qu'on ne récupère.

On a trouvera ci-après un algorithme en langage de description qui donne un aperçu sur le déroulement des tests nécessaires au déclenchement du récupérateur. Les étiquettes qui apparaissent dans le texte correspondent a celles qui sont dans l'algorithme.

Reduire :

```
début
  Si Taille restante en allocation < Besoin pour une Etape [1]
    alors Finir la fournée en cours atomiquement ;
          Amorcer une nouvelle fournée ;
          Si Taille restante en allocation < Besoin pour une Etape [2]
            alors Finir la fournée en cours atomiquement ;
                  Si Taille de la copie > Seuil Saturation [3]
                    alors Saturation [4]
                    sinon Amorcer une nouvelle fournée ;
                  Fsi
          Fsi
    sinon Si Non (Fournée en cours) [5]
            alors Si Taille restant en allocation < Seuil déclenchement [6]
                    alors Amorcer une nouvelle fournée
                  Fsi
          Fsi
  Fsi
Fin
```

Chapitre 4

Mesures.

On présente ici les résultats de l'interpréteur Prolog mis en oeuvre avec la version de MALI dotée d'un récupérateur de mémoire utilisant l'algorithme de Baker et fonctionnant de manière incrémentale.

Les valeurs sont obtenues en faisant varier, pour l'exécution d'un même programme, un paramètre lié au déclenchement du récupérateur, le seuil de déclenchement, et le rapport des vitesses de récupération et de consommation.

Présentation des tests.

Le programme utilisé pour les tests est un classique des bancs de mesures, l'inversion naïve d'une liste d'éléments.

Le prédicat *nrev* est défini de la façon suivante :

```
nrev(nil, nil) ->;
nrev(a.l1, l3) ->
        nrev(l1, l2)
        conc(l2, a.nil, l3);
```

conc étant le prédicat usuel de concaténation de deux listes.

En ce qui nous concerne, les mesures portent sur la création et l'inversion d'une liste de 450 éléments.

Afin d'observer le comportement du récupérateur plus spécifiquement, on a décidé de lier à l'exécution du programme un "boulet" qui se présente en l'occurence sous la forme d'une liste d'éléments. On peut représenter un test par le programme suivant :

```
mesure ->
        < prédicat de changement des paramètres du récupérateur >
        conslist(< taille du boulet >, l)
        conslist(450, l1)
        test(nrev(l1, l2));
```

Le prédicat *conslist(x,y)* construisant la liste *y* contenant *x* éléments, et le prédicat *test(z)* rapportant le temps d'exécution et le nombre de fournées provoqués par l'exécution de *z*.

On voit que le boulet n'intervient pas dans l'inversion de la liste de 450 éléments, mais qu'il ne fait simplement qu'augmenter le travail du récupérateur. Plus le boulet est grand plus il y a de fournées déclenchées et, à priori, plus le temps d'exécution est long (car le temps passé en récupération augmente).

On s'intéresse maintenant aux paramètres faisant l'objet de variations dans les mesures.

Le rapport récupération / consommation : coefficient reliant la vitesse de consommation à la vitesse de marquage (en fait nombre d'objets visités par le curseur Baker). Si le rapport est égal à x, supérieur ou égal à 1, à chaque demande d'allocation d'un nombre n de cellules, le nombre de pas de marquage sera égal à $n * x$. C'est le cas ou le récupérateur va plus vite que le consommateur. Si le rapport est inférieur à 1, par exemple égal à $1 / x$, il faudra consommer x cellules pour effectuer 1 pas de marquage. Dans ce cas, c'est le consommateur qui est le plus rapide.

Les rapports que l'on a choisi sont :

40, 20, 10, 6, 4, 2 : récupérateur plus rapide que le consommateur.

1 : vitesses équivalentes. Une cellule allouée implique un pas de marquage.

1/2, 1/4, 1/6, 1/10, 1/20, 1/40 : consommateur plus rapide que le récupérateur.

Le seuil de déclenchement : pourcentage d'occupation de la mémoire intervenant dans le déclenchement du récupérateur. En effet celui-ci ne pourra commencer une fournée que s'il reste moins de (100 - seuil de déclenchement) % de la mémoire pour l'allocation (cf. Les paramètres directement liés au déclenchement du récupérateur).

Les seuils de déclenchement choisis sont :

0%, 20%, 40%, 60%, 80%, 90%.

Le boulet : On l'a vu précédemment, ce boulet n'intervient pas dans le calcul proprement dit, mais augmente la quantité de mémoire "utile" et modifie donc le comportement du récupérateur. Ce boulet consiste en une liste d'un certain nombre d'éléments. On a choisi comme boulets la liste vide (c'est-à-dire une absence de boulet), une liste de 4000 éléments et une liste de 6000 éléments.

On peut calculer théoriquement la quantité de mémoire nécessaire à l'inversion naïve de 450 éléments ainsi qu'à la construction du boulet.

Pour la réalisation du boulet, la liste construite est représentée en mémoire par une liste de *construits,* le premier sous-terme de celui-ci étant un *atome* et le second une référence vers le *construit* suivant. Connaissant la taille d'un construit, on peut en déduire

la quantité de mémoire occupée par le boulet.

La résolvante correspondant à l'inversion naïve d'une liste est formée d'une suite d'éléments dont la composition est la suivante : trois *construits*, un *nuplet* d'arité 4 et une *variable*. Là encore, connaissant la taille de chacun de ces composants, il est possible de calculer la quantité de mémoire nécessaire à l'inversion naïve d'une liste de 450 éléments.

Concrètement, ceci nous donne les résultats suivants :

Inversion d'une liste de 450 éléments :

1 élément =	3 construits	(3 * (6 * 2) = 36 octets)
+	1 nuplet de 4	(1 * (6 * 4) = 24 octets)
+	1 variable	(1 * (6 * 3) = 18 octets)
		78 octets

L'inversion naïve d'une liste de 450 éléments occupe donc 450 * 78 = 35100 octets.

Construction du boulet :

1 élément = 1 construit (1 * (6 * 2) = 12 octets)

Le boulet de 4000 éléments occupe donc 4000 * 12 = 48000 octets, et le boulet de 6000 éléments occupe 6000 * 12 = 72000 octets.

Taille d'un demi-espace :

Cette taille est fixée à 120000 octets auxquels il faut enlever la consommation maximale entre deux réduire c'est-à-dire 1000 cellules donc 6000 octets. La quantité de mémoire disponible dans un demi-espace est donc 114000 octets.

Les trois boulets donnent lieu à trois résultats de mesures qu'on appelle *nrev 450*, *b4000 450*, et *b6000 450*. D'après les calculs effectués précédemment on déduit que :

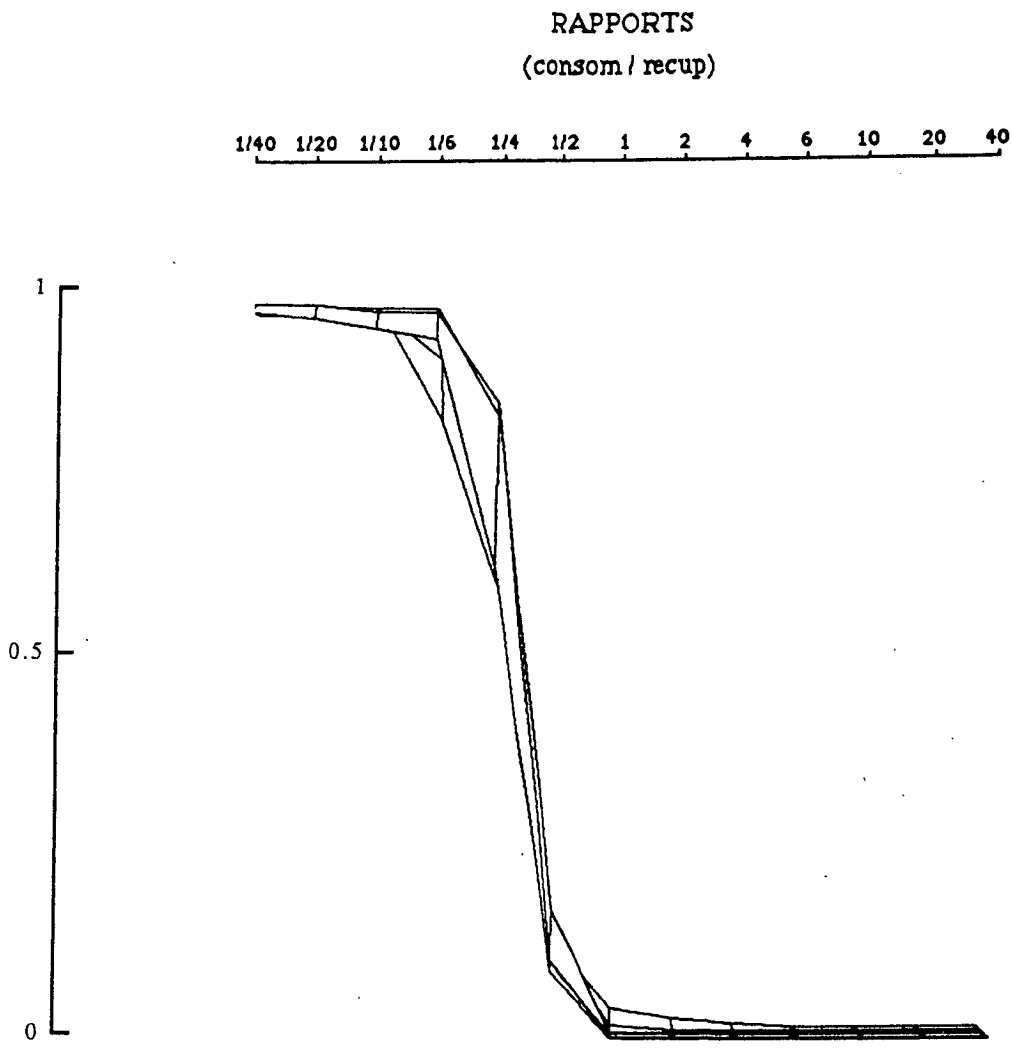
nrev 450 correspond à une occupation de la mémoire équivalente à **31 %**, *b4000 450*, à une occupation équivalente à **73 %**, et *b6000 450* occupe **94 %** de la mémoire.

Pour chaque résultat, on s'intéresse à deux fonctions du rapport et du seuil de déclenchement.

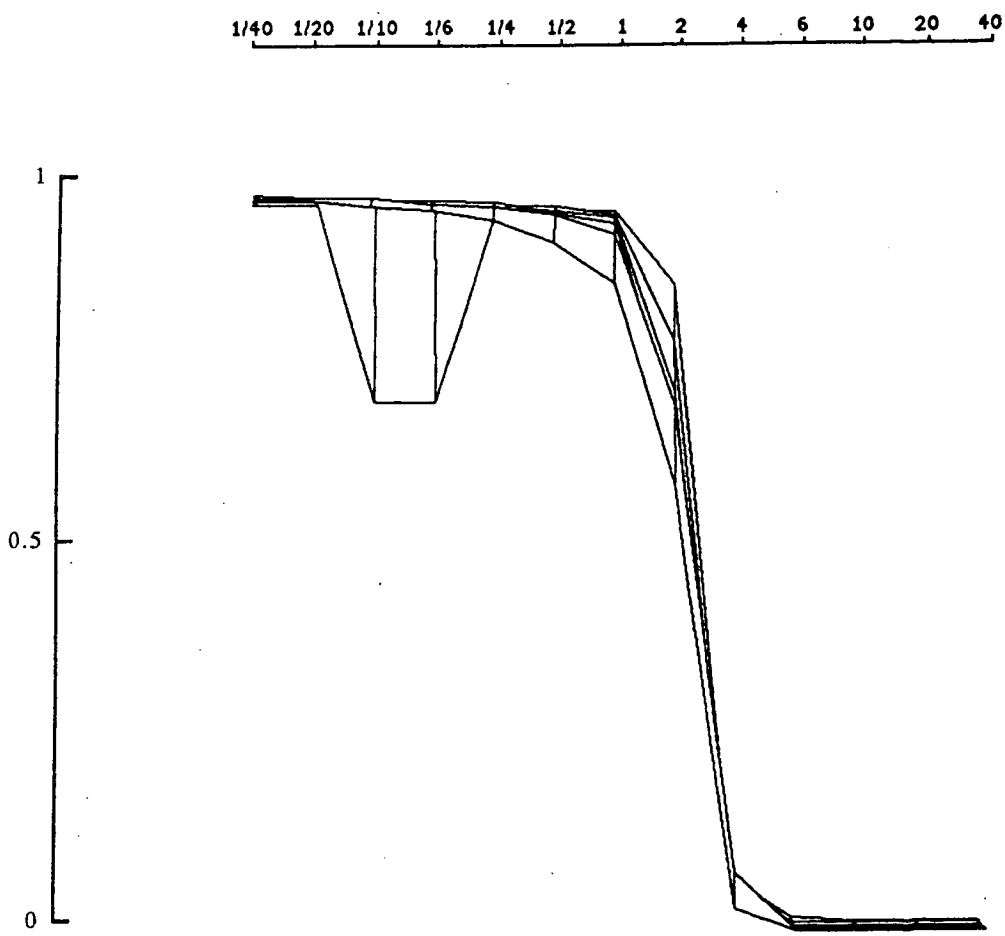
La première fonction est le *taux de saturation* qui correspond au nombre de fournées forcées sur le nombre de fournées déclenchées. Nous visualisons cette fonction sous la forme d'une projection sur le plan formé par l'axe des rapports et par celui des taux de saturation

La seconde fonction étudiée est le temps de calcul. Nouds la visualisons sous la forme d'une courbe en perspective cavalière, et sous celle d'une projection sur le plan formé de l'axe des rapports et celui des temps de calculs dans laquelle, afin d'éviter les problèmes introduits par certains pics, on décide de borner les temps par une valeur maximale choisie le plus judicieusement possible.

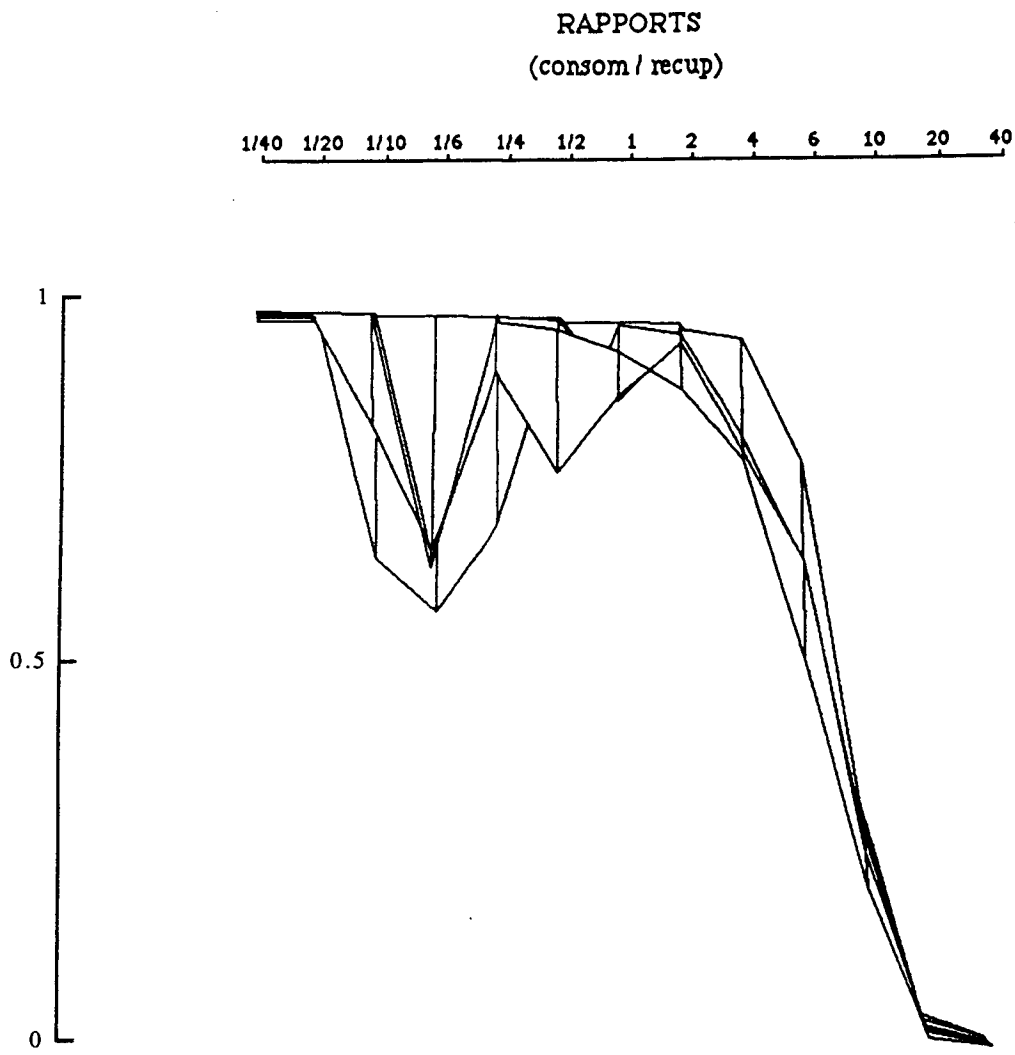
Nous dénommons les projections abaques dans la mesure où elle permettent de visualiser rapidement et simplement les résultats obtenus



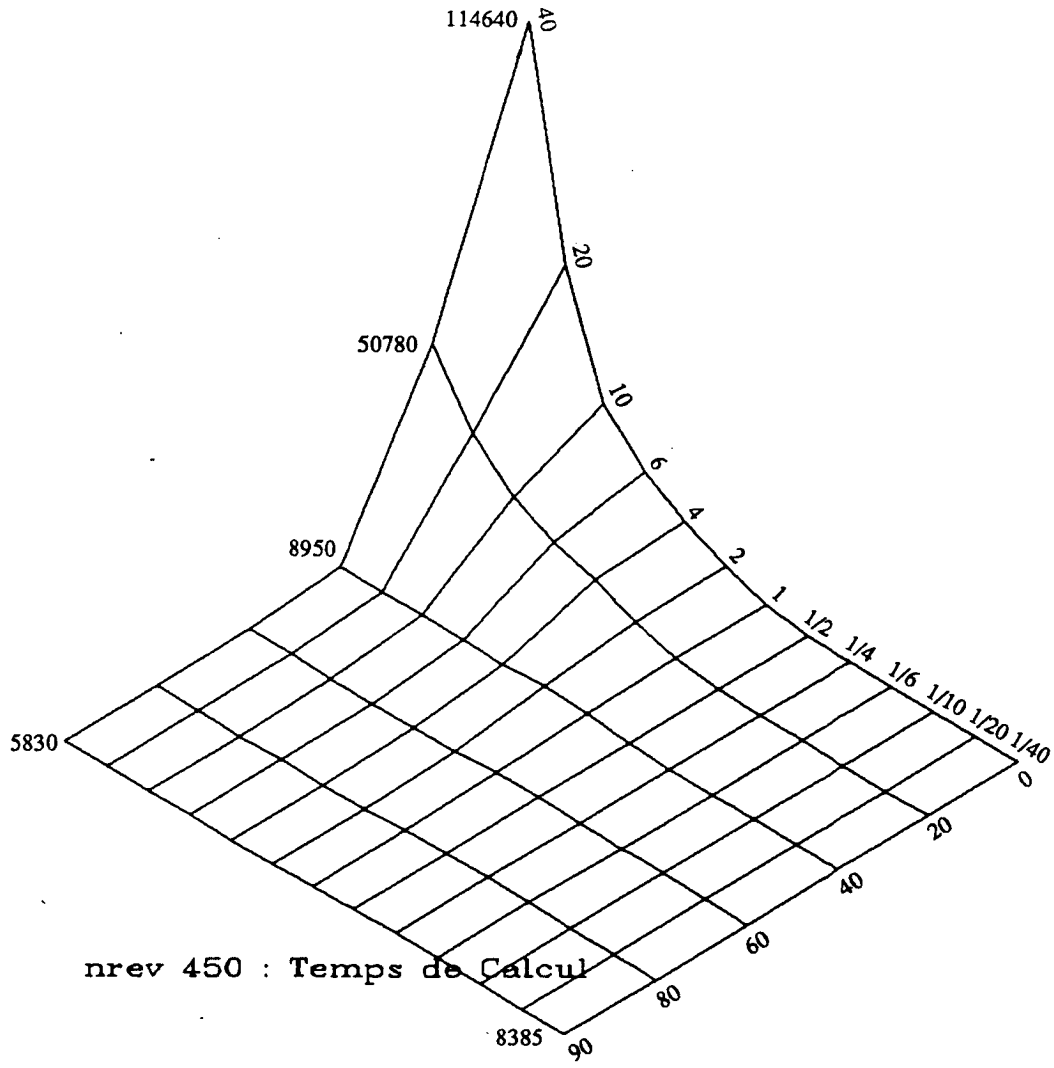
nrev 450 : Tx de Saturation (abaque)

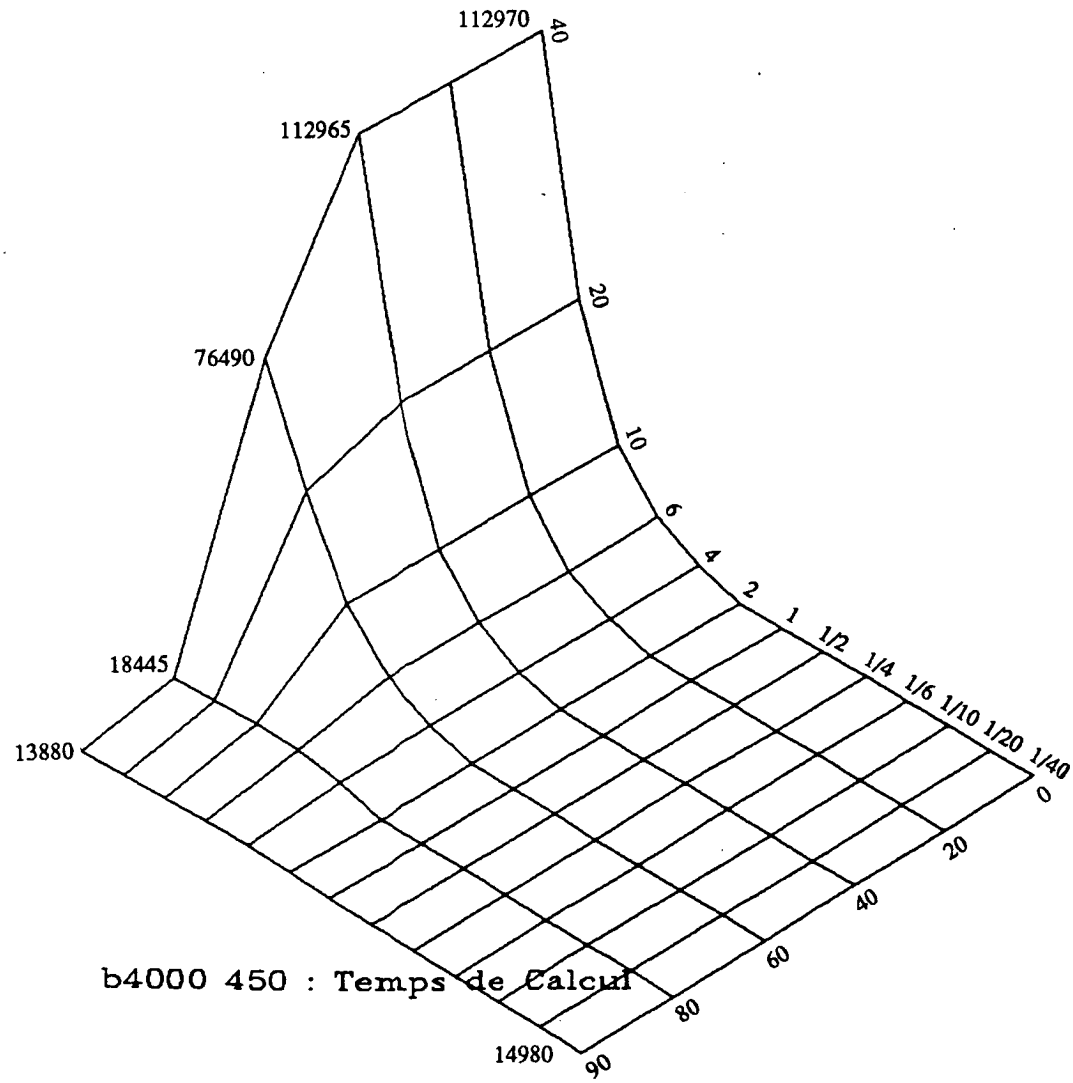


b4000 450 : Tx de Saturation (abaque)

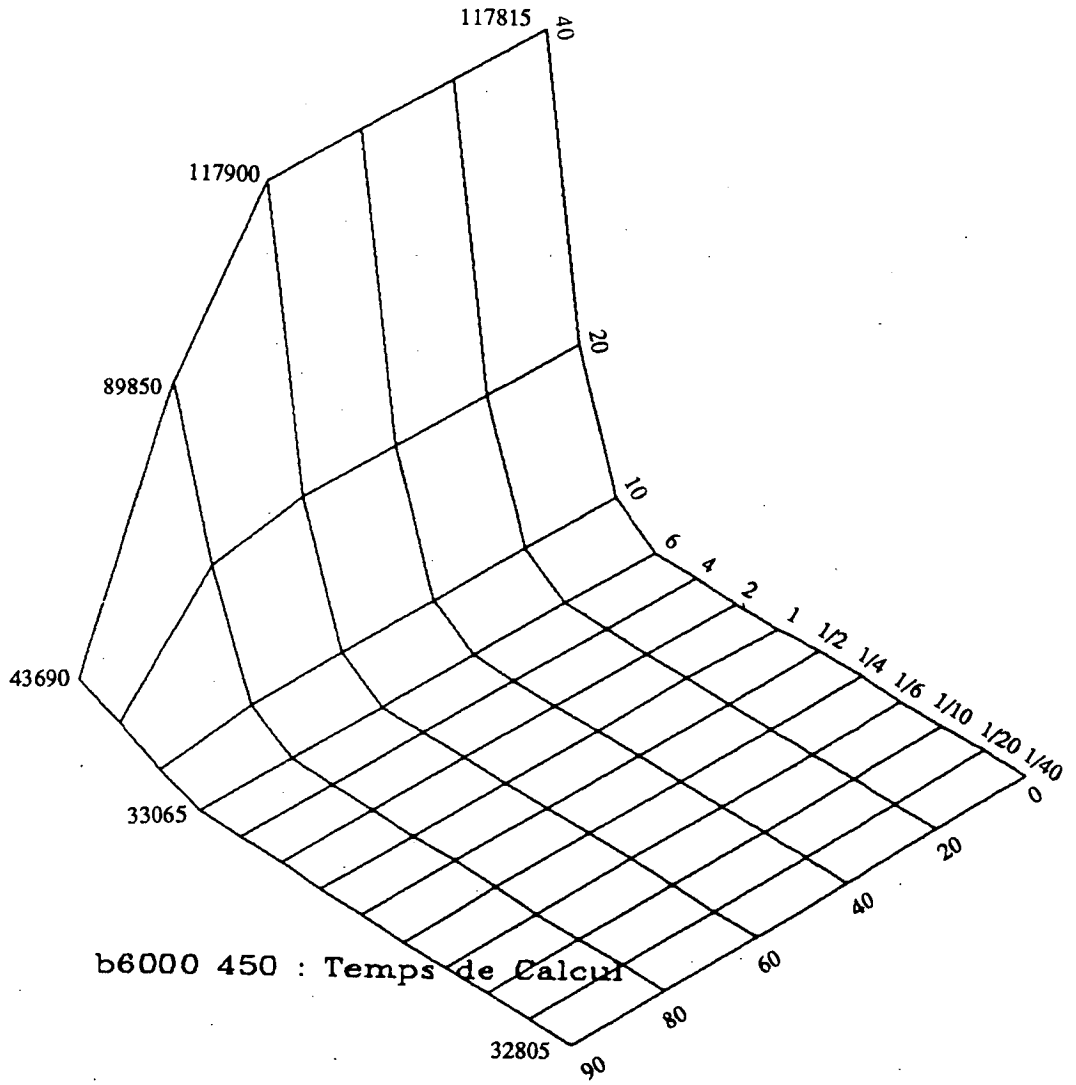


b6000 450 : Tx de Saturation (abaque)

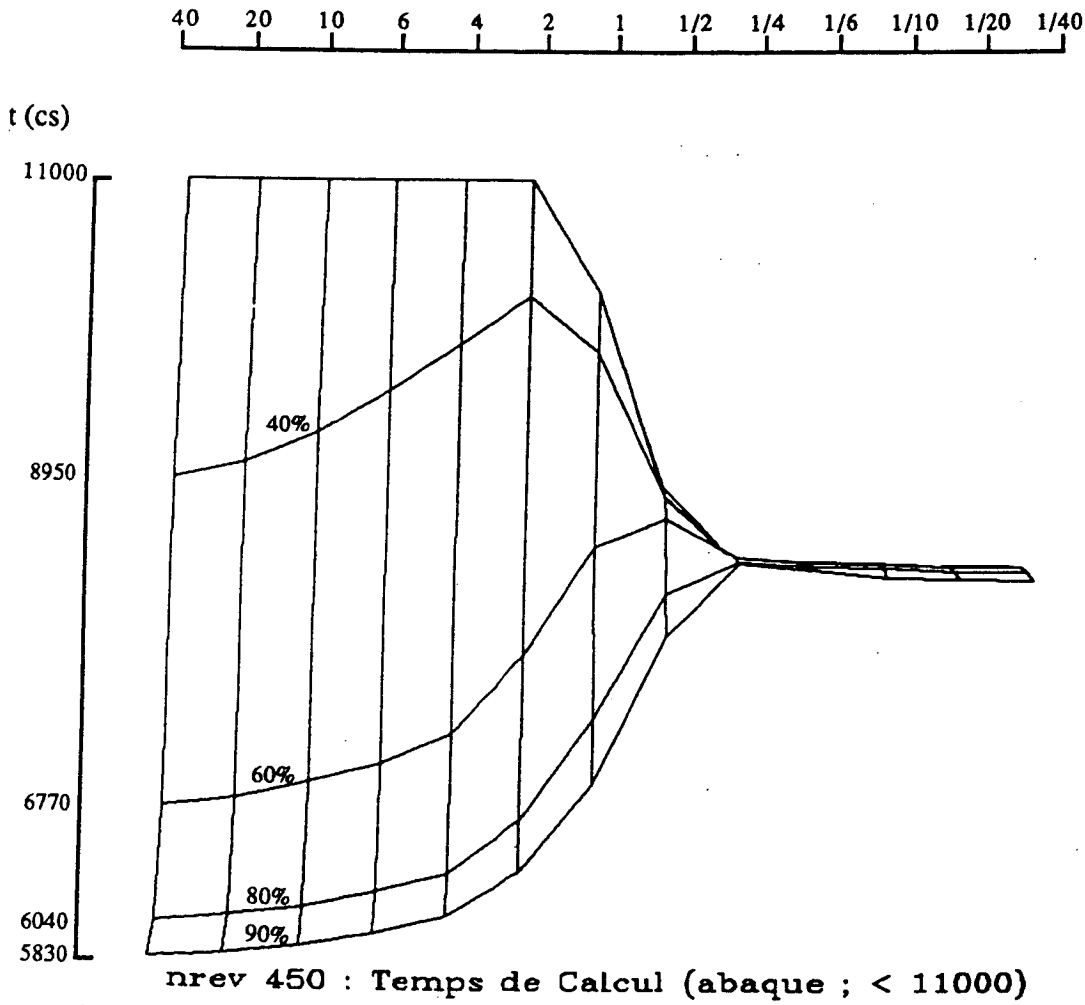

114640
50780
8950
5830
8385
40
20
10
6
4
2
1
1/2
1/4
1/6
1/10
1/20
1/40
0
20
40
60
80
90
nrev 450 : Temps de Calcul



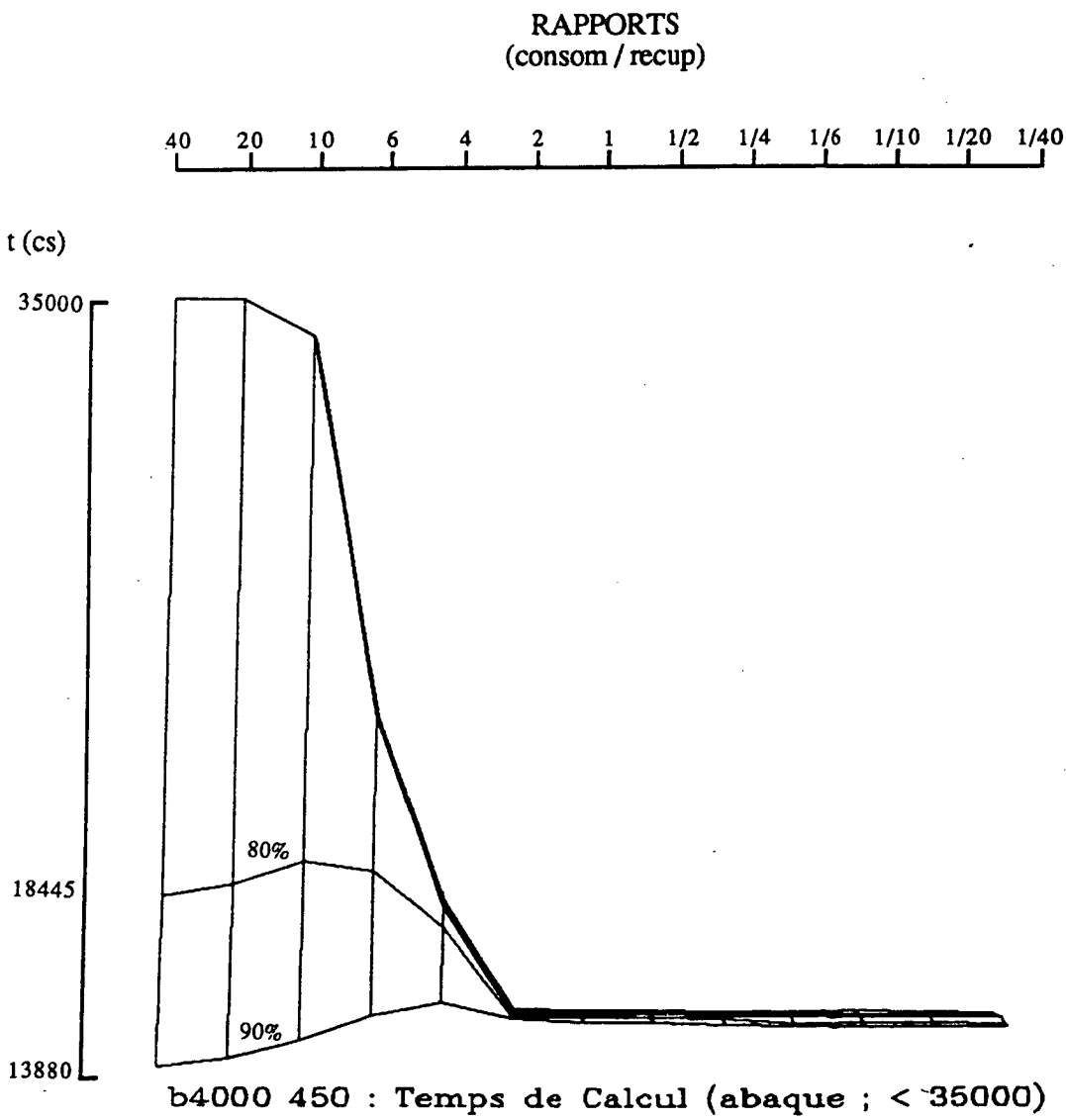
b4000 450 : Temps de Calcul


117815
117900
89850
43690
33065
32805
40
20
10
6
4
2
1
1/2
1/4
1/6
1/10
1/20
1/40
0
20
40
60
80
90
b6000 450 : Temps de Calcul

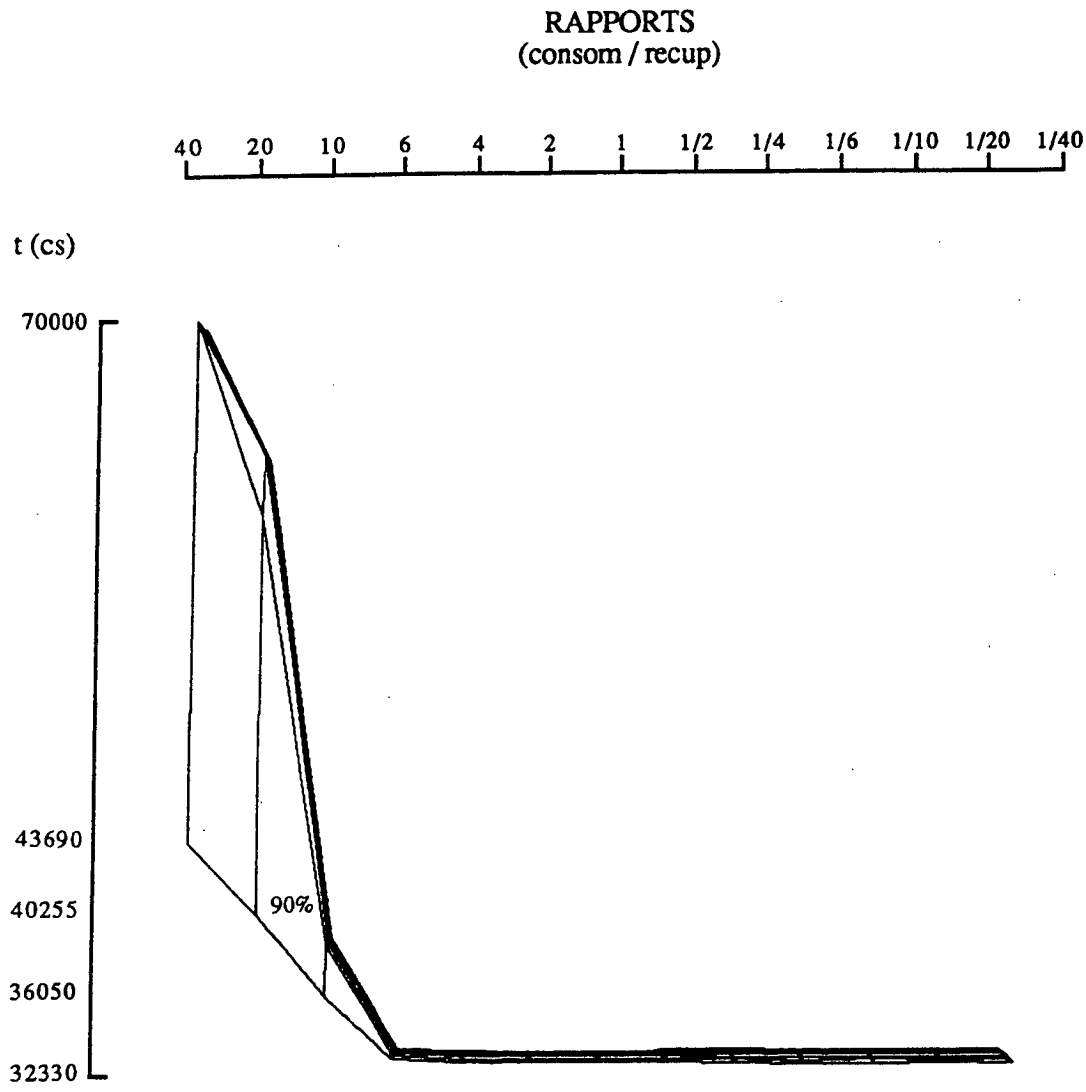
RAPPORTS
(consom / recup)



nrev 450 : Temps de Calcul (abaque ; < 11000)



b4000 450 : Temps de Calcul (abaque ; < 35000)



b6000 450 : Temps de Calcul (abaque ; < 70000)

Commentaires des résultats.

Le taux de saturation (pages 42 à 44).

Un premier résultat apparait de façon flagrante au vu des trois courbes : on dénote tout de suite l'existence d'une "marche" entre le taux de saturation à 0, c'est-à-dire le comportement se rapprochant le plus du temps-réel, et le taux de saturation proche de 1, correspondant à un état où pratiquement toutes les fournées sont forcées et se font en exclusion. On peut considérer qu'il n'y a pas de comportement intermédiaire. On appellera *zone de saturation* la zone correspondant à un taux de saturation proche de 1, et *zone de comportement temps-réel* la zone correspondant a un taux de saturation égal à 0.

Rappelons que ce qui varie dans les trois courbes est la taille du boulet. L'augmentation de cette taille déplace la "marche" vers les rapports des vitesses croissants et restreint donc la zone de comportement temps-réel.

Un autre résultat apparait, tout aussi évident : le seuil de déclenchement n'a pas d'influence sur le taux de saturation. Cette indépendance est quelque peu perturbée dans la zone de saturation quand la taille du boulet augmente, mais ces perturbations accompagnent un état dégradé du système et n'ont, à notre sens, pas d'autre signification.

Les temps de calculs (pages 45 à 50).

Les courbes en perspective cavalière (pages 45, 46, 47) font observer une zone plane dont la valeur moyenne évolue avec la taille du boulet et la présence d'un pic qui se situe approximativement aux mêmes valeurs quelque soit la taille du boulet et qui correspond toujours aux valeurs minimales du seuil de déclenchement et aux valeurs maximales du rapport des vitesses. On peut remarquer que ce pic à tendance à s'élargir quand on augmente la taille du boulet.

En ce qui concerne les abaques (pages 48, 49, 50) on retrouve la zone de saturation et la zone de comportement temps-réel. Pour la zone de saturation, on voit que ni le

rapport des vitesses, ni le seuil de déclenchement n'ont d'influence sur le temps de calcul. Par contre, dans la zone de comportement temps-réel, il apparait que le temps de calcul dépend de façon simple du seuil de déclenchement (plus le seuil est grand, plus le temps est petit), et de façon plus complexe, du rapport des vitesses.

N'oublions pas que ces courbes sont tronquées assez fortement pour éliminer le phénomène du pic et que des différences qui apparaissent nettement, seraient quasi invisibles si l'on avait maintenu toutes les valeurs.

D'autre part les meilleurs temps de calcul obtenus se situent toujours au point où le seuil de déclenchement et le rapport des vitesses sont les plus grands, dans la zone de comportement temps-réel, sauf pour le cas dégradé du boulet de 6000 éléments. Dans ce cas les temps de calcul les plus bas sont observés dans la zone de saturation.

Enfin il faut bien considérer que ces temps de calcul sont plus mauvais que ceux de la version de Baker à saturation. On donne, à titre indicatif, le meilleur temps de la version incrémentale et le temps de la version à saturation pour *nrev 450* :

Baker à saturation : 4268 cs.
Baker incrémental : 5830 cs.

Conclusion.

On a pu voir les problèmes engendrés par l'implantation dans MALI d'un récupérateur de mémoire fonctionnant suivant l'algorithme de Baker et de manière incrémentale. Ces problèmes ont pour origine, du fait même du parallélisme, la certitude qu'on doit toujours avoir de travailler sur un objet présent dans le nouvel espace.

De ce fait l'obligation de mises à jour des objets accédés, ainsi que les tests omniprésents de l'état du récupérateur laissaient présager des mauvais résultats en temps que ce système donnerait.

On doit néanmoins tenir compte que le but recherché est tout autre dans une version incrémentale. Ce but est de distribuer le temps de garbage sur toute la durée du calcul afin de ne pas bloquer l'utilisateur pendant une période consacrée à la récupération comme c'est le cas dans une version à saturation.

Références.

[BAK 78] H.G. BAKER : " List-processing in real-time on a serial computer". *Commun.* ACM 21, 4 (avril 1978).

[BEK 86] Y. BEKKERS, B. CANET, O. RIDOUX, L. UNGARO : "MALI : A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages".
Proc. of the 2nd Int. Logic programming, IEEE, Sept 1986, Salt-Lake city, USA.

[RID 86] O. RIDOUX, "Gestion de mémoire temps-réel des langages de programmation relationnelle."
Thèse, Université de Rennes I, 1986.

Partie II

Mise en oeuvre des algorithmes Lisp2 et Morris

dans le système MALI.

Michel Le Hénaff

Résumé :

La machine MALI est un système destiné à servir de noyau à la mise en œuvre des langages de programmation logique. Elle est dotée d'un récupérateur de mémoire autonome, qui dans sa version logicielle est séquentiel. Nous présentons dans ce rapport le remplacement de la technique "standard" de récupération par celle Lisp 2 et celle de Morris qui permettent une compaction de la mémoire en conservant l'ordre des structures de données utiles. Ce remplacement implique des modifications de la technique de représentation en mémoire des informations des interpréteurs des langages de programmation logique. Nous mesurons ensuite les performances en mémoire et en vitesse des deux nouvelles versions, et nous les comparons avec celles de la mise en œuvre "standard".

Mots clef :

programmation logique, mémoire, *ramasse-miettes* , marquage, compaction, mesures, performances.

Chapitre 1

PRINCIPES DES ALGORITHMES DE COMPACTION LISP 2 ET MORRIS

1.1 le ramasse-miettes

Le *ramasse-miettes* est un processus qui consiste à rassembler les portions de mémoire inutiles. Il comprend généralement deux phases distinctes :

(1) identifier les espaces inutilisés qui peuvent être récupérés ;

(2) rassembler ces espaces pour les rendre accessibles à l'utilisateur.

La phase (1) revient en général à marquer les cellules utiles ; à partir d'un ensemble de racines, on suit les pointeurs contenus dans les cellules pour marquer ainsi toutes celles qui sont accessibles.

La phase (2) peut se faire de deux façons :

(2.1) incorporation des cellules inutiles dans une liste appelée liste des libres ; cette méthode est particulièrement utilisée pour des objets allouables de taille fixe (car il n'en résulte pas de morcellement) ;

(2.2) compaction des cellules utiles dans un bout de la mémoire, l'autre bout de la mémoire formant un espace utilisable pour l'allocateur.

Les deux algorithmes que nous présentons réalisent la phase (2.2) du *ramasse-miettes* (une phase de marquage a préalablement marqué les cellules utiles).

1.2 présentation des algorithmes

Les algorithmes de Lisp 2 et de Morris permettent de compacter des cellules de taille quelconque. Les objets sont constituées par un nombre (≥ 1) de cellules contiguës.

La technique de compaction de Lisp 2 nécessite un format d'objet particulier décrit par la Figure 1. En effet la première cellule est réservé pour stocker l'information nécessaire à la mise à jour des pointeurs. Le contenu de cette cellule est une adresse : le futur emplacement de la cellule. Un objet doit aussi contenir sa taille.

L'algorithme de Morris est lui totalement ignorant de la structure des objets. Pour lui l'espace est une succession de cellules reconnaissables à leurs marques. Comme les cellules peuvent contenir des pointeurs et des données, deux bits par cellule sont nécessaires pour identifier les pointeurs, les pointeurs retroussés, les données, et les cellules inactives.

La première phase du *ramasse-miettes* marque chaque cellule active (initialement toutes les marques valent inactif). Pour le compacteur de Lisp 2 , la première cellule contient un pointeur non nul si l'objet est marqué et un pointeur nul sinon.

1.2.1 Lisp 2

La phase de marquage est déjà réalisée, il y a une valeur non nulle dans la premiére cellule de chaque objet utile.

L'algorithme, décrit par Knuth [3, Knu p. 602–603], utilise trois passes linéaires de la mémoire. La première passe a deux buts :

(1) agréger les objets non marqués adjacents ; cela pour accélérer les passes suivantes.

(2) calculer les nouvelles adresses de chaque objet marqué ; cette nouvelle adresse est la somme des tailles de tous les objets marqués précédents, et est mise dans le premier champ de la cellule.

La seconde passe met à jour les pointeurs des objets marqués en les faisant pointer sur les nouvelles adresses où les objets vont être relogés. La troisième passe recopie les objets marqués en début de mémoire.

La Figure 2 illustre les effets de chaque passe. Sur ce schéma nous n'avons représenté que les pointeurs vers un seul objet C.

1.2.2 Morris

L'algorithme de Morris [4, Mor] [5, Mor] organise la mise à jour des pointeurs grâce à l'idée suivante : soient les cellules A, B, et C dont les contenus pointent sur la cellule Z, et Z a pour contenu X (Figure 3(a)). Lors des visites successives de A, B, et C on peut représenter la situation initiale sans perdre d'information en construisant une liste linéaire de ces cellules dont la racine est Z, et où le contenu de Z est sauvegardé dans

le dernier élément de la liste (Figure 3(b)). Puis lorsque l'on arrive à Z, à un moment où le nouvel emplacement Z' où le contenu de Z va être relogé est connu, on rétablit la situation initiale en mettant à jour les pointeurs (Figure 3(c)).

L'algorithme de Morris nécessite une marque de deux bits par cellule. Ces bits indiquent la nature du contenu avec la convention suivante :

0 : inactif ;
1 : non pointeur (donnée) ;
2 : pointeur ;
3 : pointeur retroussé.

Pour cet algorithme, l'espace mémoire est considéré comme une succession de cellules banalisées ; ainsi il n'oblige pas les pointeurs à référencer le début des objets.

La phase de marquage a positionné les marques de toutes les cellules de la mémoire. Elle a aussi calculé le nombre de cellules actives. Ce nombre permet, lors des passes de l'algorithme de Morris , de calculer les nouvelles positions des cellules actives.

La compaction s'effectue par trois parcours linéaires de la mémoire. Deux passes pour la mise à jour des pointeurs : la première dans une direction, la seconde dans l'autre direction ; la troisième passe relogeant les cellules en bout de mémoire. Si la première passe est effectuée dans le sens de la compaction alors la phase de recopie des cellules peut être combinée avec la seconde phase de mise à jour (les cellules progressant dans le sens opposé à cette passe).

Pour compacter les cellules en début de mémoire, la première passe partira donc de la fin de l'espace vers le début. Elle met à jour les seuls pointeurs vers l'arrière, ainsi que ceux référençant leur propre cellule. La seconde passe est dirigée dans l'autre sens : elle part du début de l'espace vers la fin. Elle met à jour les pointeurs vers l'avant et reloge les cellules à leur nouvel emplacement en début d'espace.

Le cœur de l'algorithme de Morris est la mise à jour des pointeurs dans la direction de la passe par la technique suivante : si les emplacements A, B, C pointent sur Z qui a pour contenu X, pendant le parcours nous avons construit la liste Z, C, B, A en les liant par des pointeurs, X étant placé en A. Pour ce faire il y a eu une modification de l'information, cela est indiqué par les valeurs appropriées des marques (pointeur retroussé). L'opération de mise à jour se fait lors du passage sur Z en parcourant la liste, et en faisant A, B, C pointer sur le nouvel emplacement Z' où le contenu de Z sera relogé. Ainsi, avec une passe dans chaque direction, tous les pointeurs sont mis à jour.

La Figure 4 présente les différentes phases de l'algorithme de Morris :

situation initiale : Figure 4(a) ;

passe 1 : de la fin de l'espace vers le début, mise à jour des pointeurs en arrière :

Figure 4(b) en arrivant à C la liste des cellules pointant sur C a été construite ;

Figure 4(c) connaissant la nouvelle position de C, la mise à jour des pointeurs est effectuée ;

passe 2 : du début de l'espace vers la fin, mise à jour des pointeurs en avant :

Figure 4(d) on arrive en C, les cellules précédentes ont été recopiées en début d'espace et les cellules pointant sur C sont liées dans une liste ;

Figure 4(e) les pointeurs vers C sont mis à jour ;

situation finale : Figure 4(f) la cellule C est relogée à son nouvel emplacement, le reste de la passe tassant les cellules suivantes.

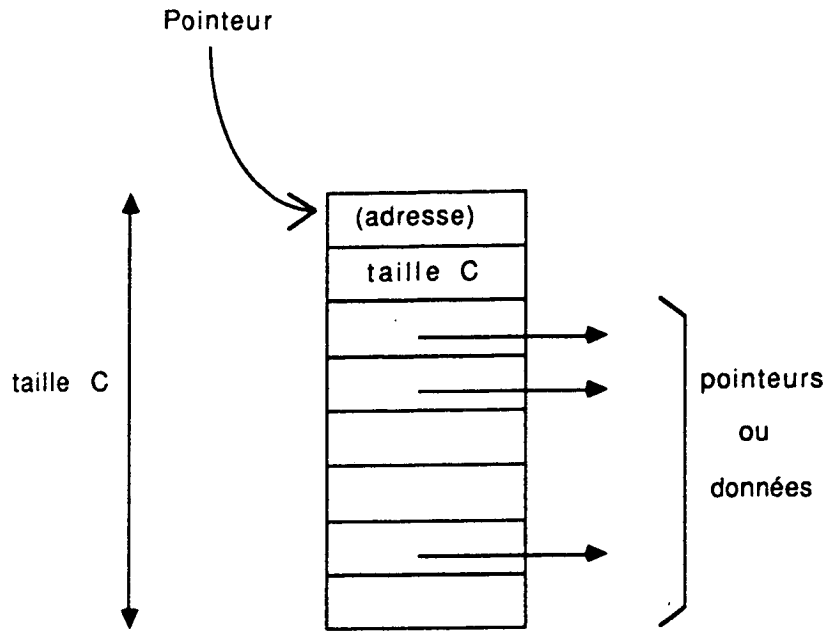


Figure 1 : format d'un objet pour Lisp 2

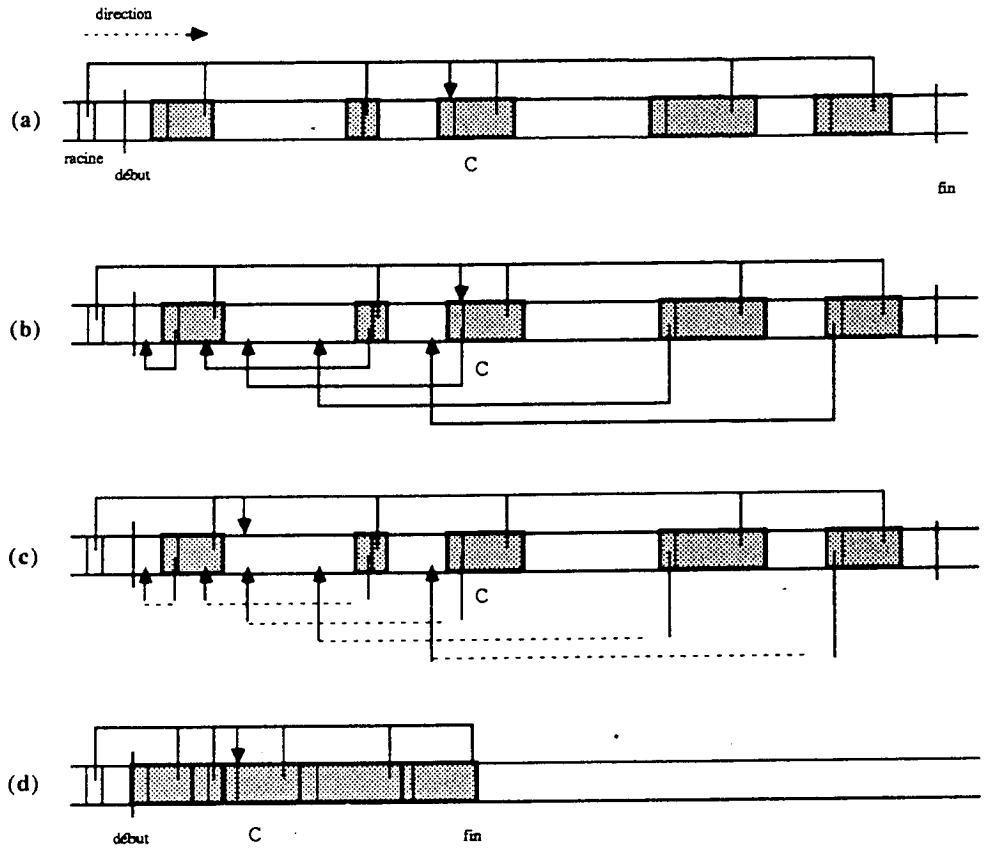


Figure 2 : schéma décrivant les effets de l'algorithme de Lisp 2

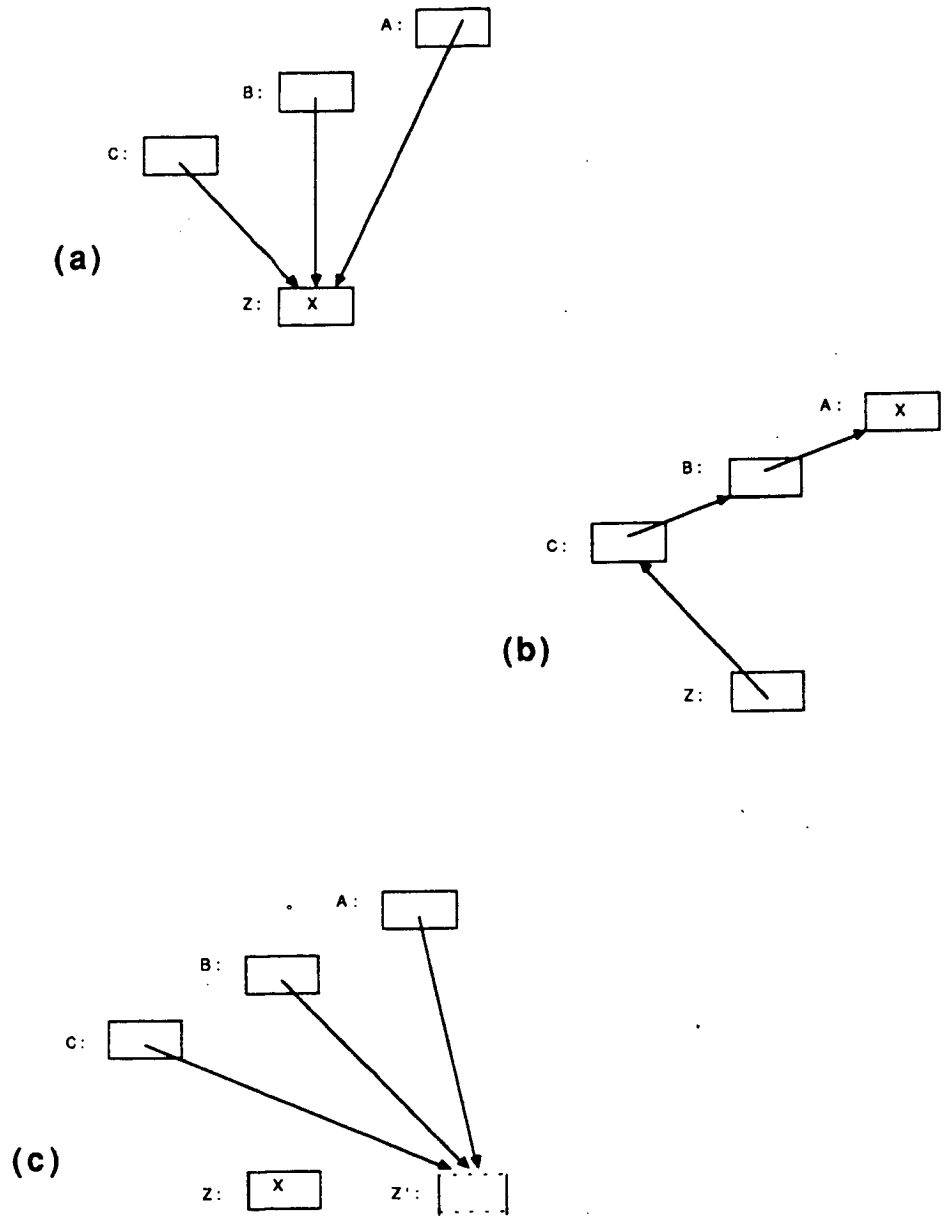


Figure 3 : principe de l'algorithme de Morris

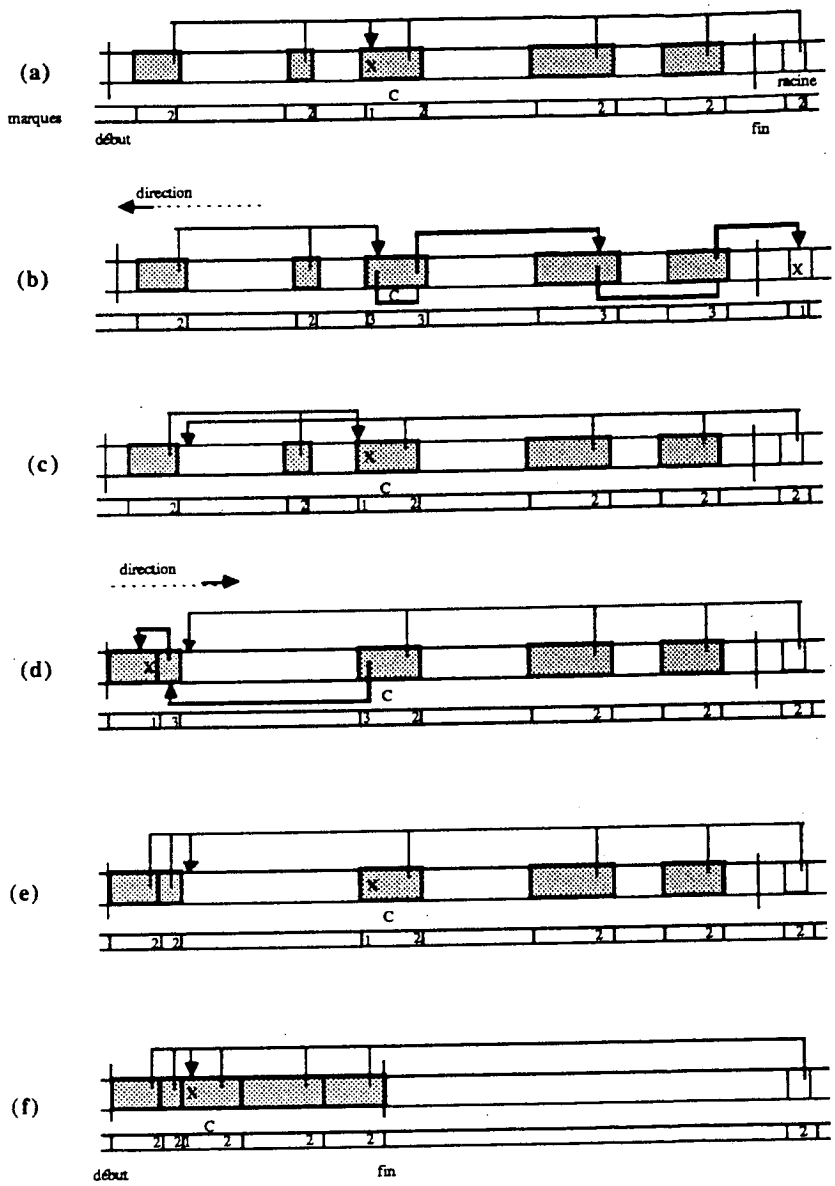


Figure 4 : schéma décrivant les effets de l'algorithme de Morris

Chapitre 2

MISE EN ŒUVRE DES ALGORITHMES LISP 2 ET MORRIS DANS LE SYSTEME MALI

2.1 MALI

MALI [2, Bek] (Machine Adaptée aux Langages Indéterministes) est une machine abstraite dont la sémantique est appropriée à la mise en œuvre des langages de programmation logique. Sa principale caractéristique est la définition d'une politique d'utilisation de la mémoire et la gestion automatique de cette mémoire par un récupérateur autonome. Il existe des implantations logicielles et matérielles de cette machine. Dans sa version logicielle, MALI comporte plusieurs couches dont celles décrivant la technique de représentation en mémoire des informations des interpréteurs des langages de programmation logique, et celles décrivant le récupérateur de mémoire.

MALI a donc deux activités :

Les services rendus à l'utilisateur : la construction et le parcours de termes représentant l'état d'un interpréteur ; ainsi que des commandes spécifiques permettant d'obtenir l'effet du non-déterminisme ;

La récupération automatique et optimale de la mémoire occupée par les termes devenus inutiles à la représentation de l'état d'interprétation.

L'une des originalités du système MALI est d'utiliser une seule mémoire spécialisée où sont représentés les termes et les résolvantes.

2.1.1 Représentation des informations

Dans MALI , toutes les informations sont représentées par des structures appelées

cellules . Ces *cellules* peuvent contenir des *Noms* de *Termes* ; le *Nom* donne la *Nature* et la *Sorte* du *Terme* et permet d'accéder à ce *Terme* .

Les *Natures* possibles sont :

Atome ,
Construit ,
Nuplet ,
Variable ,
Variable à attribut ,
Niveau .

La *Sorte* indique l'arité des *Nuplets* ou donne un typage élémentaire aux *Atomes* , *Construits* , *Variables* , et *Variables à attribut* .

Les structures de données

Un *Nom* d'*Atome* contient directement sa propre valeur.

Un *Nom* de *Construit* permet d'accéder à son sous-terme gauche et à son sous-terme droit.

Un *Nom* de *Nuplet* d'arité n donne accès à la représentation de ses n sous-termes.

Les variables

Les *Variables* et les *Variables à attribut* de MALI correspondent aux variables des langages de programmation logique ; ainsi elles peuvent être liées à un *Nom* de *Terme* . Tout *Nom* qui désigne la variable avant la substitution, désigne le *Terme* après.

En plus de la *Variable* , la *Variable à attribut* possède une valeur associée (seulement dans son état libre) : un *Terme* appelé *Attribut* .

2.1.2 La création et le parcours de termes

Les commandes de MALI offrent à l'utilisateur les moyens de construire des termes et d'y accéder.

Pour ce groupe de commandes, MALI offre les piles nécessaires. Ainsi, l'utilisateur n'a pas à s'occuper de ces piles, MALI les loge dans son espace mémoire et les gére elle même.

2.1.3 Le contrôle de la recherche

C'est pour mieux connaitre l'espace utile que MALI prend en charge la gestion de la pile de recherche (i.e. installation et suppression de points de reprise, création et liaison de variables et retour-arrière). Les fonctions de MALI réalisent ainsi la stratégie de recherche en profondeur d'abord.

La pile de recherche est représentée par une liste de *Termes* nommés *Niveau* qui sont des structures donnant accès à la représentation des *Termes* sauvegardés. Pour sauvegarder un *Terme* , on utilise sa représentation courante : c'est ce qu'on appelle le "partage OU". Ainsi, puisque les substitutions de variables provoquent des effets de bord sur la représentation des *Termes* , il faut restaurer cette représentation en annulant les effets de bord. C'est le rôle de la "traînée" : chaque *Niveau* de la pile de sauvegarde donne accès à la liste des modifications apportées à la représentation.

Ainsi l'empilement d'un niveau dans la pile de recherche correspond à la mise en place d'un point de reprise, le dépilement d'un *Niveau* correspond à un retour-arrière. La coupure de Prolog est réalisée en forçant la pile de recherche à valoir le *Niveau* voulu.

2.1.4 Le contrôle de la récupération mémoire

Pour effectuer la récupération mémoire, le récupérateur à besoin de connaitre les *Noms* des termes utiles. Pour cela il y a synchronisation entre l'utilisateur et le récupérateur par des appels réguliers au récupérateur en fournissant à chaque fois les germes de la récupération.

Ces appels sont nommés "réduire" et déclenchent le récupérateur si l'occupation de la mémoire atteint un certain seuil.

L'utilisateur doit indiquer le maximum d'espace qu'il peut consommer entre deux "réduire". Le seuil de déclechement est alors inférieur ou égal à la taille de la mémoire moins la quantité fournit par l'utilisateur.

La saturation est annoncée lorsque, suite à une récupération, l'occupation de la mémoire atteint un seuil appelé seuil de saturation. Ce seuil permet d'empêcher une trop forte augmentation des temps d'interprétation. Lors de l'approche de la saturation, les récupérations devenant systématiques à chaque appel de "réduire", les performances de vitesse se dégradent.

Les trois paramètres contrôlant la récupération mémoire sont donc :

- la consommation maximale de mémoire entre deux "réduire" ;
- le seuil de déclenchement ;
- le seuil de saturation.

Dans MALI ces deux seuils sont égaux et valent la taille de la mémoire diminuée de la consommation maximale entre deux "réduire".

2.1.5 La logique d'utilité

Pour les langages comme Prolog, l'évolution des termes est contrainte par la stratégie de recherche. Le parcours de détection des termes utiles doit donc interpréter la représentation des variables en fonction de la pile de recherche et de la traînée. Ceci est appelé la logique

d'utilité [6, Rid] spécifique aux langages sans affectation pratiquant une recherche en profondeur d'abord.

Le parcours de la représentation utile est effectué en partant des *Niveaux* les plus récents vers les *Niveaux* les plus anciens. Pour chaque élément de la pile de recherche, il y a marquage des termes sauvegardés et parcours de la partie de traînée correspondant à ce *Niveau* . Ce parcours permet de détecter que la liaison d'une variable est inutile, ou qu'une variable est inutile alors que sa valeur de liaison est utile.

La libération anticipée de variable :

Lors de l'examen d'un élément de traînée, si aucun terme sauvegardé plus récent que cet élément de traînée ne fait référence à la variable, alors la substitution de la variable est inutile. Aucun terme sauvegardé n'utilise cette variable dans sa situtation liée, nous la délions. Sa valeur de liaison est détruite si elle n'est plus utilisée.

La dévariabilisation :

Si tous les termes sauvergardés dont la représentation utilise une variable sont plus récents que l'élément de traînée qui la cite, alors cette variable est inutile. Le récupérateur court-circuite alors cette variable en remplaçant ses références par sa valeur de liaison.

La détection de la dévariabilisation s'effectue en comparant le niveau de création de la variable et le niveau correspondant à l'élément de traînée qui cite la variable. Si les deux niveaux sont les mêmes alors aucun terme n'utilise cette variable dans son état libre. La détection d'une dévariabilisation est réalisée à deux moments :

(1) lors de la substitution des variables ;

(2) lors de la récupération mémoire au moment du parcours de la pile de recherche et de la traînée (car la pile de recherche a pu être modifiée par d'éventuelles coupures).

2.1.6 La récupération mémoire

Dans MALI , les *Termes* , la pile de recherche et la traînée occupent de la mémoire. Le rôle du récupérateur de mémoire est de détecter les représentations inutiles de termes, de *Niveaux* de la pile de recherche et de la traînée.

Une représentation peut devenir inutile pour plusieurs raisons :

- Perte d'accès si la représentation n'est pas un sous-terme des germes de la récupération ou si elle ne sert pas pour un *Terme* sauvegardé.
- Perte d'accès lors de la modification de la pile de recherche par les actions des coupures et du retour-arriere.
- Application de la logique d'utilité avec libération anticicipée de variable et dévariabilisation.

La détection des représentations inutiles demande donc un parcours de la représentation des germes de la récupération et des termes sauvegardés, mais aussi une interprétation de l'état des variables en fonction de la pile de recherche et de la traînée.

La procédure de récupération à partir de l'algorithme de Baker

Après avoir présenté les principes de la récupération, intéressons-nous à sa mise en œuvre dans le système MALI .

Dans la version logicielle de MALI , le récupérateur se déclenche lors de la saturation de la mémoire, la procédure de *ramasse-miettes* s'effectue alors totalement.

L'algorithme de *ramasse-miettes* actuellement utilisé dans MALI est celui de Baker [1, Bak]. Son principe est le suivant :
il divise la mémoire en deux demi-espaces.
Les passes de marquage et de tassage sont réunies en une seule phase.
Lorsque l'espace d'allocation arrive à saturation, les objets utiles sont recopiés dans l'autre demi-espace. Les objets déjà recopiés servent de pile de parcours. Lorsque le parcours est terminé, toutes les représentations utiles ont été relogées dans l'espace d'arrivée avec leurs pointeurs mis à jour.

Cette procédure utilise la logique d'utilité pour faire de la dévariabilisation et de la libération anticipée de variables. Ainsi, si lors du parcours des termes utiles, elle rencontre la citation d'une variable déjà détectée dévariabilisable, cette citation est remplacée par la valeur de liaison de la variable.

Avec l'algorithme de Baker , il y a une seule phase qui effectue le marquage et la recopie des objets. La dévariabilisation n'est pas faite sur le champ : elle nécéssite deux récupérations.

Au sujet de l'algorithme de Baker , nous pouvons déjà avancer les appréciations suivantes :
la méthode de Baker a pour qualités :

1. de nécessiter une seule phase effectuant le parcours et la recopie des termes utiles, au lieu d'une phase de marquage et deux ou trois parcours linéaires de la mémoire pour les autres *ramasse-miettes* ;

2. de se passer de pile de parcours, les termes déjà recopiés servant de pile ;

Le point (1) explique que la procédure de récupération est de complexité temporelle linéaire avec le volume de mémoire utile.

Par contre l'algorithme de Baker a pour défauts :

1. la diminution de moitié de l'espace d'allocation car la mémoire est divisée en deux ;

2. la modification de l'ordre initial des termes après une récupération.

2.2 Implantation de Lisp 2

2.2.1 Représentation des informations

La cellule

Pour expliquer la façon dont est mis en œuvre l'algorithme de Lisp 2 dans le système MALI , nous allons maintenant préciser la représentation des termes dans la mémoire spécialisée de MALI .

Tous les termes nécessaires à la représentation de l'état d'un interpréteur sont construits à partir d'objets. Ces objets ont les six natures possibles déjà vues : *Atome* , *Construit* , *Nuplet* , *Variable* , *Variable à attribut* et *Niveau* .

Tous les objets sont représentés par des cellules de base. La structure d'une cellule est la suivante :
elle est composée d'un champ *indicateur* et d'un champ *information* :

Si une cellule contient un *Nom* de terme, alors la signification des deux champs de la cellule est la suivante :
l'*indicateur* donne la *Nature* et la *Sorte* du terme, le champ *information* contient la valeur du terme pour un *Atome* ou, pour un autre terme, la référence de sa représentation.

Un *Construit* ou un *Nuplet* est représenté par des cellules contenant les *Noms* de leurs sous-termes. Un atome est représenté par une cellule contenant une valeur. Une *Variable* est représentée par des cellules dont une cellule contient soit l'indication "non liée", soit le *Nom* du terme substitut. Les *Variables à attribut* ont une représentation identique aux *Variables* , avec en plus une cellule donnant accès à son attribut. Un *Niveau* est représenté par des cellules permettant d'accéder aux termes sauvegardés.

La mise en œuvre de Lisp 2 dans le récupérateur nous a conduit à modifier la représentation des objets. Ces modifications sont importantes car elles changent la taille des termes et ainsi influencent les performances de MALI . Nous présentons maintenant ces transformations qui sont au nombre de trois :

(1) ajout d'une cellule supplémentaire par objet ;

(2) suppression de la cellule donnant le niveau de naissance dans les *Variables* et les *Variables à attribut* ;

(3) suppression des déports dans les *Variables à attribut* et les *Niveaux* .

1ère modification de la représentation des objets :

Ajout d'une cellule supplémentaire à la représentation de chaque objet (sauf pour l'*Atome*) pour obtenir la structure nécessaire à la mise en œuvre de Lisp 2 (voir

Figure 1). L'*Atome* ne nécessite pas de cellule supplémentaire car il contient directement une valeur.

Cette cellule supplémentaire est mise en début d'objet. Son utilisation est la suivante :

l'indicateur : il permet de stocker l'*indicateur* de l'objet dans sa représentation elle même. Comme nous connaissons la structure de la représentation d'un objet à partir de son *indicateur*, celui-ci nous donne la taille de l'objet (nous en aurons besoin dans le récupérateur pour la phase de tassage) ;

le champ *information* : il peut contenir une adresse ; ce champ servira de marque pour chaque objet, ainsi que d'emplacement pour contenir la nouvelle adresse de l'objet.

Le résultat de cette modification est donc une cellule supplémentaire pour les *Construits*, les *Nuplets*, les *Variables*, les *Variables à attribut* et les *Niveaux*.

2nde modification de la représentation des objets :

Elle concerne les *Variables* et les *Variables à attribut*, et consiste à supprimer, dans leurs représentations, la cellule contenant le niveau de naissance de la variable.

Pour détecter la dévariabilisation des variables, nous avons besoin de connaître le niveau de naissance de la variable ainsi que le niveau où est liée cette variable. Le niveau de substitution est donné par la pile de recherche et la traînée. Dans la version de MALI utilisant Baker, le niveau de naissance est mis dans la représentation des variables car l'algorithme de tassage modifie l'ordre des objets dans la mémoire.

Avec le récupérateur basé sur l'algorithme Lisp 2, l'ordre de création des objets est conservé. Dans ce cas, le niveau de naissance d'une variable est le niveau dont la représentation précède immédiatement la représentation de cette variable.

Nous obtenons donc une économie d'une cellule par *Variable* et *Variable à attribut*.

3ème modification de la représentation des objets :

Elle transforme la représentation des *Variables à attribut* et des *Niveaux*. Ces objets ne contenaient pas directement le *Nom* de l'*Attribut* pour la *Variable à attribut* ou les *Noms* des termes sauvegardés pour le *Niveau*. Nous y trouvions seulement un déport vers ces *Noms*. C'était pour que le marquage ne soit pas au courant de de la structure des objets qu'il traverse.

Maintenant, une *Variable à attribut* contient le *Nom* de l'attribut et un *Niveau* contient les *Noms* des termes sauvegardés.

Cette modification entraîne donc un gain d'une cellule pour la représentation des *Variables à attribut* et des *Niveaux*.

En résumé :

La comparaison de la représentation des informations suivant l'utilisation de l'algorithme de Baker ou de Lisp 2 donne le résultat suivant :

pour les *Construits* et les *Nuplets* , leurs représentations nécessitent une cellule supplémentaire avec Lisp 2 ;

pour les *Atomes* , les *Variables* et les *Niveaux* , leurs représentations occupent la même place pour les deux versions ;

pour les *Variables à attribut* , nous les représentons avec une cellule de moins dans la version Lisp 2 .

2.2.2 Gestion de la mémoire

L'algorithme de Lisp 2 travaille sur un seul espace et conserve l'ordre des cellules. Les conséquences, sur la gestion de la mémoire, de son utilisation dans le récupérateur sont les suivantes :

Un espace réunifié : Nous avons toujours un seul espace où sont représentés tous les termes ; mais dans la version utilisant Baker cet espace est divisé en deux. Avec Lisp 2 il n'y a plus cette division et ainsi l'espace d'allocation se retrouve deux fois plus grand.

La gestion du retour-arrière simplifié : Lisp 2 conservant l'ordre de création des termes, il permet une mise en œuvre du retour-arrière par gestion en pile de l'espace d'allocation :
lors du retour-arrière nous pouvons systématiquement faire retomber le début de la zone d'allocation jusqu'au dernier point de reprise (i.e. le niveau sommet) de la pile de recherche ; nous récupérons ainsi la place de tous les objets créés depuis ce point de reprise (cela n'est pas toujours possible avec la version Baker).

2.2.3 La création et le parcours de termes

Nous avons trouvé, dans certaines commandes de parcours de termes, une utilisation de la cellule supplémentaire ajoutée à la représentation des objets.

En dehors de ces commandes et de la récupération mémoire, le contenu de cette cellule supplémentaire n'est pas significatif.

A l'occasion d'une comparaison ou d'un parcours en parallèle de deux termes, la cellule supplémentaire des objets parcourus sert de pile de parcours. Elle contient alors le chaînage vers le père de l'objet dans le parcours. Dans ces commandes, nous nous passons ainsi d'allocation d'espace pour réaliser les piles de parcours.

Toutes les autres piles sont logées en fin de mémoire et progressent vers le bas (dans la version séquentielle de Baker toutes les piles sont logées dans l'ancien espace d'allocation).

2.2.4 Le récupérateur de mémoire

La dévariabilisation

La détection d'une dévariabilisation s'effectue en comparant l'emplacement de la mémoire représentant la variable et l'emplacement de la mémoire représentant le niveau de la pile de recherche où la variable a été liée. Si le second précède le premier alors nous pouvons en déduire qu'aucun terme ne verra cette variable libre après un retour-arrière ; elle est donc dévariabilisable.

Le marquage d'un terme

Nous décrivons ici le marquage des objets utiles à partir d'une racine donnée. Le marquage d'un terme consiste à parcourir ce terme pour marquer à utile tous les sous-termes accessibles.

Nous disposons d'une marque par objet dans la cellule supplémentaire rajoutée en début de chaque objet. Initialement, tous les objets de la mémoire ont la marque inutile dans le champ *information* de cette cellule.

Lors du marquage d'un terme, la pile de parcours est mise dans la cellule supplémentaire des objets parcourus. Si un objet est dans la pile, la première cellule contient alors dans le champ *information* le chaînage vers le père de l'objet ; son *indicateur* est modifié et nous donne le prochain composant de l'objet à parcourir. Au dépilement d'un objet, le chaînage reste dans le champ *information* de la première cellule, l'*indicateur* est restauré à sa valeur initiale.

Cette méthode réalise bien le marquage. En effet, si la première cellule d'un objet contient une adresse alors cet objet est déjà marqué, sinon la première cellule contient toujours la marque inutile.

De plus lors du marquage d'un terme, nous court-circuitons les références à une variable dévariabilisée : sa référence est remplacée par la valeur de liaison. La variable dévariabilisée n'est pas marquée.

La phase de marquage

La phase complète de marquage comporte plusieurs parties :

1. le prémarquage à inutile de tous les objets de la mémoire ;
2. le marquage des germes de la fournée de récupération ;
3. le parcours de la pile de recherche et de la traînée avec :

   marquage des termes sauvegardés à chaque point de reprise,
   dévariabilisation,
   libération anticipée de variables.

Le prémarquage :

Il consiste en un parcours linéaire de la mémoire en marquant les objets à inutile : le champ *information* de la première cellule de chaque objet recoit alors une valeur

spéciale indiquant que l'objet n'est pas marqué. Le passage d'un objet au suivant s'effectue facilement à l'aide de l'*indicateur* de la première cellule de l'objet qui permet d'en obtenir la taille.

Le parcours de la pile de recherche et de la traînée :
Il a pour rôle de marquer les termes sauvegardés de chaque niveau de cette pile, et de traiter les éléments de la traînée.
Deux cas se présentent pendant ce traitement :

soit la variable concernée est marquée.
Dans ce cas, si la variable est dévariabilisable alors la mémoire la représentant est démarquée. La valeur de liaison qui, elle, est utile sera recopiée à la place de chaque référence à la variable pendant la phase de tassage.

soit la variable n'est pas marquée.
Dans ce cas aucun terme n'utilise cette variable dans sa situation liée. Nous délions alors la variable, sa valeur de liaison ne sera donc jamais marquée par l'intermédiaire de la variable.

La phase de tassage

Cette phase de tassage est réalisée à partir de l'algorithme de Lisp 2 . Les seuls aménagements apportés à cet algorithme consistent lors de la passe de mise à jour des références à :

1. court-circuiter les dernières références à des variables dévariabilisées ;
2. tirer parti de notre connaissance de la structure des objets, à partir de l'*indicateur* de l'objet qui se trouve dans sa première cellule, pour accéler la mise à jour des références.

La compaction s'effectue donc en trois parcours linéaires de la mémoire (voir Figure 2).

$1^{ère}$ passe :
Elle calcule les nouvelles positions des objets utiles. La future adresse d'un objet est la somme des tailles des objets utiles qui le précèdent. Cette nouvelle adresse est mise dans la première cellule de l'objet.

Cette passe agrège aussi les objets inutiles en vue d'accélérer les deux passes suivantes.

2^{nde} passe :
Elle met à jour tous les pointeurs à l'aide des nouvelles adresses mise en place par la passe précédente.

C'est à ce moment là que sont traitées les références à une variable dévariabilisée. Lors de la mise à jour d'une référence à une variable, la représentation de cette variable peut ne pas être marquée. Nous nous trouvons alors face à une variable dévariabilisée au moment du parcours de la pile de recherche et de la traînée pendant la phase de marquage. La valeur de liaison est alors recopiée à la place de la référence à la variable et ensuite est mise à jour.

3ème passe :

Elle reloge tout simplement les objets utiles en les tassant vers le début de l'espace.

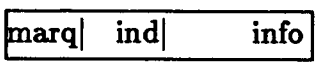
2.3 Implantation de Morris

2.3.1 Représentation des informations

La cellule

Dans la mémoire spécialisée de MALI , les termes sont représentés à l'aide d'objets, eux-mêmes constitués de cellules.

L'utilisation de l'algorithme de Morris oblige à modifier la structure de ces cellules pour y incorporer une marque. La cellule a donc dans cette version trois champs : le champ marque, le champ *indicateur* et le champ *information* .



Une marque est donc attachée à chaque cellule, à la différence de la version avec Lisp 2 où il y a une seule marque par objet. Cela permet alors d'appliquer le tassage à la Morris qui est totalement ignorant de la structure des objets.

La mise en œuvre de Morris a aussi amené des modifications dans la représentation des objets. Ces transformations, comme avec Lisp 2 , découle du fait que le compactage utilisant Morris conserve l'ordre des objets dans la mémoire.

Par rapport à la version de MALI utilisant Baker , il y a deux transformations dans la représentation des objets :

(1) suppression de la cellule donnant le niveau de naissance des *Variables* et *Variables à attribut* ;

(2) suppression des déports dans les *Variables à attribut* et les *Niveaux* .

Ces modifications sont les mêmes que celles effectuées lors de la mise en œuvre de Lisp 2 . Nous allons les décrire rapidement.

$1^{ère}$ modification de la représentation des objets :

Elle consiste à supprimer le champ niveau de naissance dans les *Variables* et les *Variables à attribut* . L'ordre des objets étant toujours le même après les récupérations, le niveau de naissance d'une variable sera le premier niveau dont la représentation précède la variable dans la mémoire.

2^{nde} modification de la représentation des objets :

Elle supprime le déport dans les *Variables à attribut* et les *Niveaux* .

Une *Variable à attribut* contient maintenant, entre autres, le *Nom* de l'attribut. Un *Niveau* contient les *Noms* des termes sauvegardés.

Conclusion :

Si nous comparons la représentation des objets entre la version de MALI utilisant Baker et celle utilisant Morris , nous obtenons le résultat suivant :

les *Construits* , les *Nuplets* et les *Atomes* occupent la même place dans les deux versions ;

les *Variables* et les *Niveaux* sont représentés avec une cellule de moins dans la version Morris ;

les *Variables à attribut* sont représentées avec deux cellules de moins.

Nous pouvons déduire de cela que la représentation d'un même état de l'interpréteur sera plus volumineuse dans la version basée sur Baker que dans celle basée sur Morris .

2.3.2 La gestion de la mémoire

Elle est la même que dans la version avec Lisp 2 , c'est-à-dire :

L'espace d'allocation : Les deux espaces de Baker sont réunifiés pour en former un seul.

Le retour-arrière : Il est mis en œuvre à la manière d'une pile, avec libération de l'espace jusqu'à l'emplacement du niveau sommet.

2.3.3 La création et le parcours de termes

Dans cette version, toutes les piles servant à la création et au parcours de termes sont logées en fin de mémoire et progressent vers le bas.

2.3.4 Le récupérateur de mémoire

La dévariabilisation

Le critère de dévariabilisation est le même que celui décrit dans le récupérateur de mémoire de Lisp 2 . Il consiste en une comparaison du niveau de naissance de la variable (donné par la position de la variable dans la mémoire) avec le niveau de substitution de la variable.

Le marquage d'un terme

Le marquage d'un terme consiste ici à positionner la marque de chaque cellule accessible suivant son contenu.

Lors de ce marquage, le nombre de cellules actives est aussi calculé, cela pour déterminer les nouvels emplacements des cellules actives.

Nous disposons de deux bits de marque par cellule, les quatres valeurs possibles pour une marque ont les significations suivantes :

1. inactive ;

2. donnée ;

3. pointeur ;

4. marque temporaire.

Le parcours d'un terme pour le marquer n'utilise pas de mémoire supplémentaire pour la pile de parcours. Nous avons réalisé cette pile par la méthode de retroussement des pointeurs.

L'algorithme suivant décrit le marquage de termes en ne considérant que des *Atomes* et des *Nuplets* .

```
var pointeur_sur_cellule sommet, pionnier, ref;
    indicateur ind;
debut
sommet := 0;
faire
  ind := pionnier->indicateur;
  cas (ind)
      atome:
         pionnier->marque:=donnee;
         progresser(pionnier);
      nuplet:
         pionnier->marque := pointeur;
         ref := pionnier->refcell;
         si (ref->marque <> inactive)
           alors
              progresser(pionnier);
           sinon
              empiler(pionnier);
              pour i:=1 a (taille_nuplet(ind) - 1) faire
                  ref->marque := temporaire;
                  ref := ref +1;
              fait
              pionnier := ref;
         fsi
  fcas
jusqu'a (sommet = 0)
fin
```

```
proc progresser (var pointeur_sur_cellule: reference)
var booleen recommencer;
debut
  recommencer := vrai;
  tant_que (recommencer = vrai) faire
      reference := reference - 1;
      si ((reference->marque = temporaire) ou (sommet = 0))
        alors recommencer := faux;
        sinon
              reference := reference + 1;
              depiler(reference);
      fsi
  fait
fin


proc empiler( pointeur_sur_cellule: reference )
debut
  reference->refcell := sommet;
  sommet := reference;
fin

proc depiler( var pointeur_sur_cellule: reference)
var pointeur_sur_cellule pere;
debut
  pere := sommet->refcell;
  sommet->refcell := reference;
  reference := sommet;
  sommet := pere;
fin
```

La phase de marquage

Elle commence par un parcours linéaire de la mémoire pour marquer toutes les cellules à inactif.

Puis, les germes de la récupération, fournis par l'utilisateur de MALI , sont marqués.

Viens ensuite le parcours de la pile de recherche et de la traînée qui marque les termes sauvegardés de chaque point de reprise. Pendant ce parcours, la dévariabilisation et la libération anticipée de variable est effectuée de la même manière que dans la version Lisp 2 de MALI .

La phase de tassage

Elles est réalisée par deux parcours linéaires de l'espace :
le premier de la fin de l'espace vers le début ;

le second du début vers la fin.

1ère passe :

Elle met à jour les références en arrière, et agrège les cellules inactives pour accélérer la passe suivante.
De plus, toutes les références à des variables détectées dévariabilisables sont court-circuitées. Devant une cellule marquée à pointeur, si la référence pointe sur une cellule non marquée, alors nous nous retrouvons face à la représentation d'une détectée dévariabilisable au moment de la phase de marquage. La valeur de liaison est alors recopiée à la place de la référence à la variable.

2nde passe :

Elle met à jour les références en avant et recopie les cellules actives en les compactant en début de mémoire.

Chapitre 3

MESURES COMPARATIVES DE PERFORMANCES

3.1 Présentation des trois problèmes de test

Pour évaluer les performances des différentes versions de MALI , nous utilisons l'interpréteur PrologIIR qui est construit au-dessus de MALI .

Les mesures proviennent de l'exécution des trois programmes suivants :

1. construction d'une liste ;
2. inversion naïve d'une liste ;
3. calcul des nombres premiers.

Construction d'une liste

Ce problème a été choisi pour sa simplicité et pour son temps d'exécution qui est linéaire par rapport à la taille de la liste construite. Cette seconde caractéristique nous permettra plus tard d'analyser plus précisément les comportements des trois récupérateurs.

```
conslist(n,l) ->
   val(add(n,1),n1)
   conslist(n1,n1.l);
```

L'appel conslist(0,nil) construit la plus grande liste possible et aboutit à la saturation de la mémoire. Nous mesurons le temps d'exécution à chaque fois que nous avons ajouté 500 éléments à la liste.

Inversion naïve d'une liste

Ce programme est classique pour les comparaisons de performances.

```
nrev(nil,nil) ->;
nrev(a.l1,l3) ->
   nrev(l1,l2)
   conc(l2,a.nil,l3);
```

Nous mesurons à chaque fois le temps de création et d'inversion de la liste.

Calcul des nombres premiers

Ce problème a été retenu dans les bancs de mesures car il utilise le "geler" de PrologII qui, dans l'interpréteur PrologIIR, est mis en œuvre à l'aide des *Variables à attribut* .

```
" Programme de recherche de nombres premiers (crible d'Eratosthene) "
"                                                                   "
" Au fur et a mesure du calcul, des cribles s'intercalent "
" entre le producteur d'entiers (list-ent)                "
" et le consommateur de nombres premiers (prem)           "

   multiple(x,f) ->
      val(mod(x,f),0);

   list-ent(x,x.l) ->
      val(add(x,1),x1)
      list-ent(x1,l);

   crible(f,x.l,l1) ->
      multiple(x,f)
      /
      freeze(l, crible(f,l,l1));
   crible(f,x.l,x.l1) ->
      freeze(l, crible(f,l,l1));

   prem(x.l,i) ->
      val(add(i,1),i1)
      freeze(l, crible(x,l,l1))
      freeze(l1, prem(l1,i1));

   premier ->
      freeze(l, prem(l,0))
```

```
list-ent(2,1);
```

Les mesures du temps d'exécution sont fait en fonction du nombre de nombres premiers calculés.

3.2 Performances mémoires

A partir des trois tests ci-dessus, nous allons pouvoir évaluer les performances mémoires des trois versions de MALI : Baker , Lisp 2 et Morris . Nous étudierons, ensuite, les effets des versions paquées Baker et Lisp 2 sur les performances.

3.2.1 Comparaison des trois récupérateurs

A partir de notre connaissance de la représentation des termes et des résolvantes, nous allons, dans un premier temps, calculer statiquement pour chaque version la plus grande taille de liste constructible et inversible, ainsi que la limite du calcul des nombres premiers. Nous vérifierons ensuite dynamiquement ces résultats.

Dans les trois versions considérées, la cellule est implantée sur 6 octets.

Calculs statiques

Pour tous les problèmes, la taille de la mémoire spécialisée de MALI sera spécifiée.

Dans la version Baker , la mémoire de MALI est divisée en deux ; l'espace d'allocation est alors deux fois plus petit.

Il faut de plus enlever à chaque espace d'allocation le seuil de déclenchement du récupérateur qui vaut 1000 cellules ; nous obtenons ainsi la taille effective de l'espace utilisable.

Nous donnons, ci-dessous, la taille en cellules de la représentation des objets pour les trois versions :

	baker	lisp 2	morris
Construit	2	3	2
Variable	3	3	2
Variable à attribut	5	4	3
Nuplet d'arité n	n	n + 1	n

La construction d'une liste de valeurs

Dans ce problème, nous allouons à MALI un espace mémoire de 1M octets.

La liste construite est représentée en mémoire par une liste de *Construits* . Le premier sous-terme de chaque *Construit* est un *Atome* , le second est la référence du *Construit* suivant.

nous obtenons ainsi le tableau donnant la taille de la plus grande liste constructible pour chaque version.

	baker	lisp 2	morris
taille *Construit* (en octets)	12	18	12
taille espace utile (en octets)	494000	994000	994000
taille max liste (en éléments)	41166	55222	82833

Inversion naïve d'une liste de valeurs

L'espace mémoire de MALI est ici de 240000 octets.

La résolvante dans l'exécution de ce programme est constituée d'une suite d'éléments dont le format est le suivant : un élément est formé de trois *Construits* , d'un *Nuplet* d'arité 4 et d'une *Variable* .

nous pouvons ainsi construire le tableau donnant la taille de la plus grande liste inversible naïvement.

	baker	lisp 2	morris
taille un élément (en octets)	78	102	72
taille espace utile (en octets)	114000	234000	234000
taille max liste (en éléments)	1461	2294	3250

calcul des nombres premiers

L'espace mémoire alloué à MALI est dans ce problème de 240000 octets.

Pour ce programme là, un élément de la résolvante est constitué de : deux *Nuplets* d'arité 4, deux *Variables* , une *Variable à attribut* et un *Construit* .

Nous pouvons à l'aide de ceci obtenir le nombre maximum de nombres premiers que nous pouvons obtenir dans chaque version.

	baker	lisp 2	morris
taille un élément (en octets)	126	138	102
taille espace utile (en octets)	114000	234000	234000
nombre max de nombres premiers	904	1695	2294

Vérifications dynamiques

Ces vérifications ont été effectuées en exécutant ces trois programmes sur chaque version de MALI . Elles permettent de déterminer les limites effectives des versions pour les trois problèmes.

Pour chaque problème, nous présentons les courbes des trois versions.

La figure 5 donne, pour la construction d'une liste, le temps d'exécution en ms/taille de la liste danx une mémoire de 1000000 octets ;

La figure 6 donne, pour l'inversion naïve d'une liste, le temps d'exécution en ms/taille de la liste dans une mémoire de 240000 octets ;

La figure 7 présente, pour le calcul des premiers nombres premiers, le temps d'exécution en ms/nombre de nombres premiers calculés dans une mémoire de 240000 octets.

Les résultats obtenus sont trés proches de ceux prévus par les calculs effectués précédemment.

	Baker (estimation) (mesure réelle)	Lisp 2 (estimation) (mesure réelle)	Morris (estimation) (mesure réelle)
construction d'une liste	41166 41000	55222 55000	82833 82500
inversion naïve	1461 1400	2294 2200	3250 2900
nombres premiers	904 900	1695 1650	2294 2250

Les performances mémoires sont les suivantes :

la version Morris a plus de deux fois la capacité mémoire de la version Baker ;

la version Lisp 2 se situe entre la version Baker et la version Morris .

3.2.2 Les effets de la version paquée

Il existe une version de MALI où la cellule est représentée sur 4 octets : elle est appelée version "paquée".

Nous allons ici étudier les effets, sur les versions Baker et Lisp 2 , de cette représentation des cellules.

L'examen du problème de construction d'une liste suffira à nous donner les indications nécessaire sur les effets du compactage des cellules.

Calculs statiques

L'espace mémoire alloué à MALI est de 240000 octets.

La représentation des cellules sur 4 octets (au lieu de 6) entraîne une modification de la taille des termes et de la taille de l'espace utilisable (car le seuil de déclenchement du récupérateur est exprimé en nombre de cellules).

Un simple calcul laisse présager amélioration de 50% des performances mémoires en passant des versions non paquées aux versions paquées.

Vérifications dynamiques

Les résultats des calculs précédents sont confirmés par l'exécution du programme de construction d'une liste.

la figure 8 présente les courbes des versions Baker non-paquée et Baker paquée.

La figure 9 présente les courbes des versions Lisp 2 non-paquée et Lisp 2 paquée.

Il apparaît sur les deux figures que les versions paquées et non paquées ont le même comportement général. De plus la taille de la liste construite par chaque version paquée est supérieure de 50% à celle de la liste construite par la version non paquée correspondante.

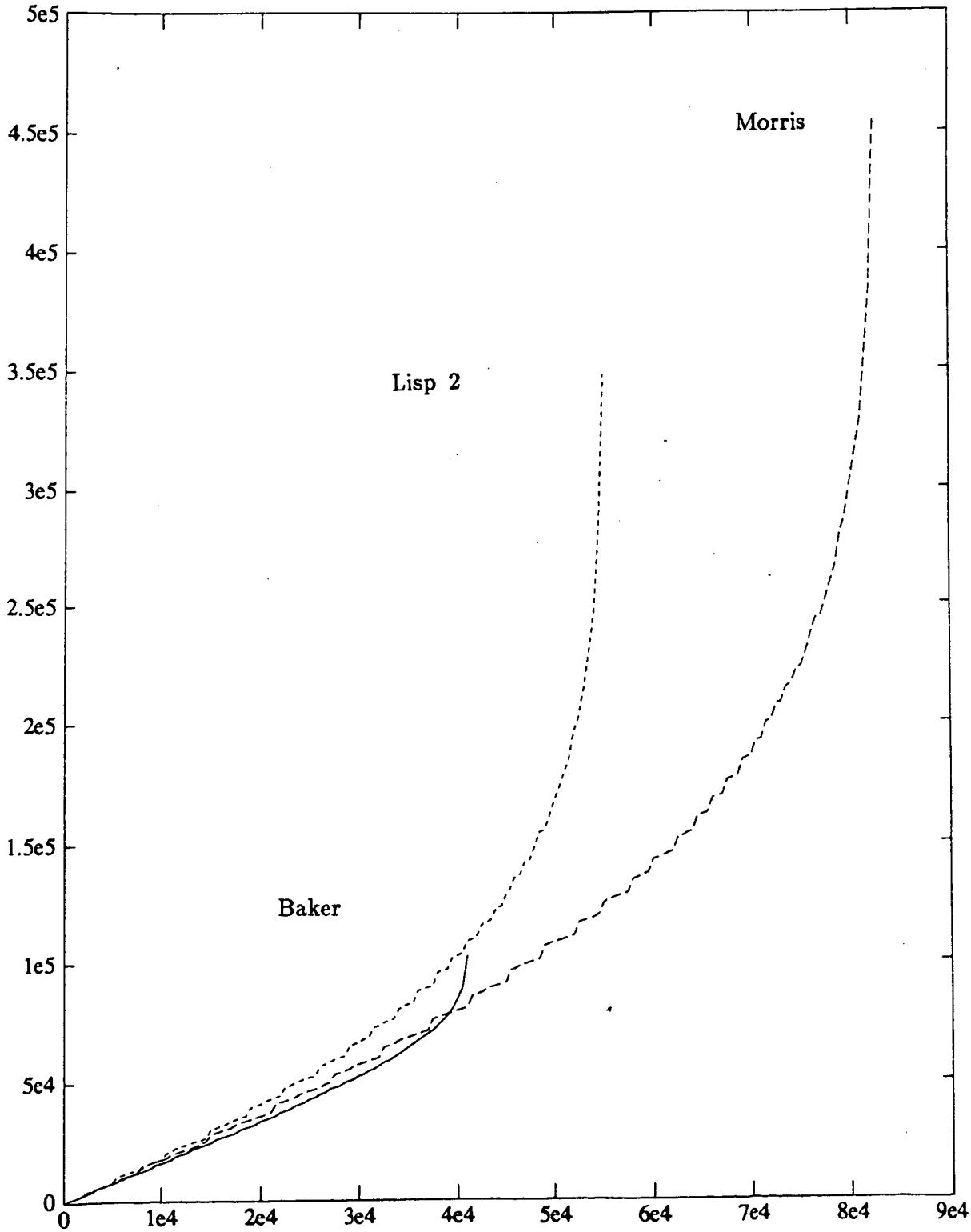


Figure 5 : construction d'une liste avec 1Mo de mémoire versions Baker , Lisp 2 et Morris

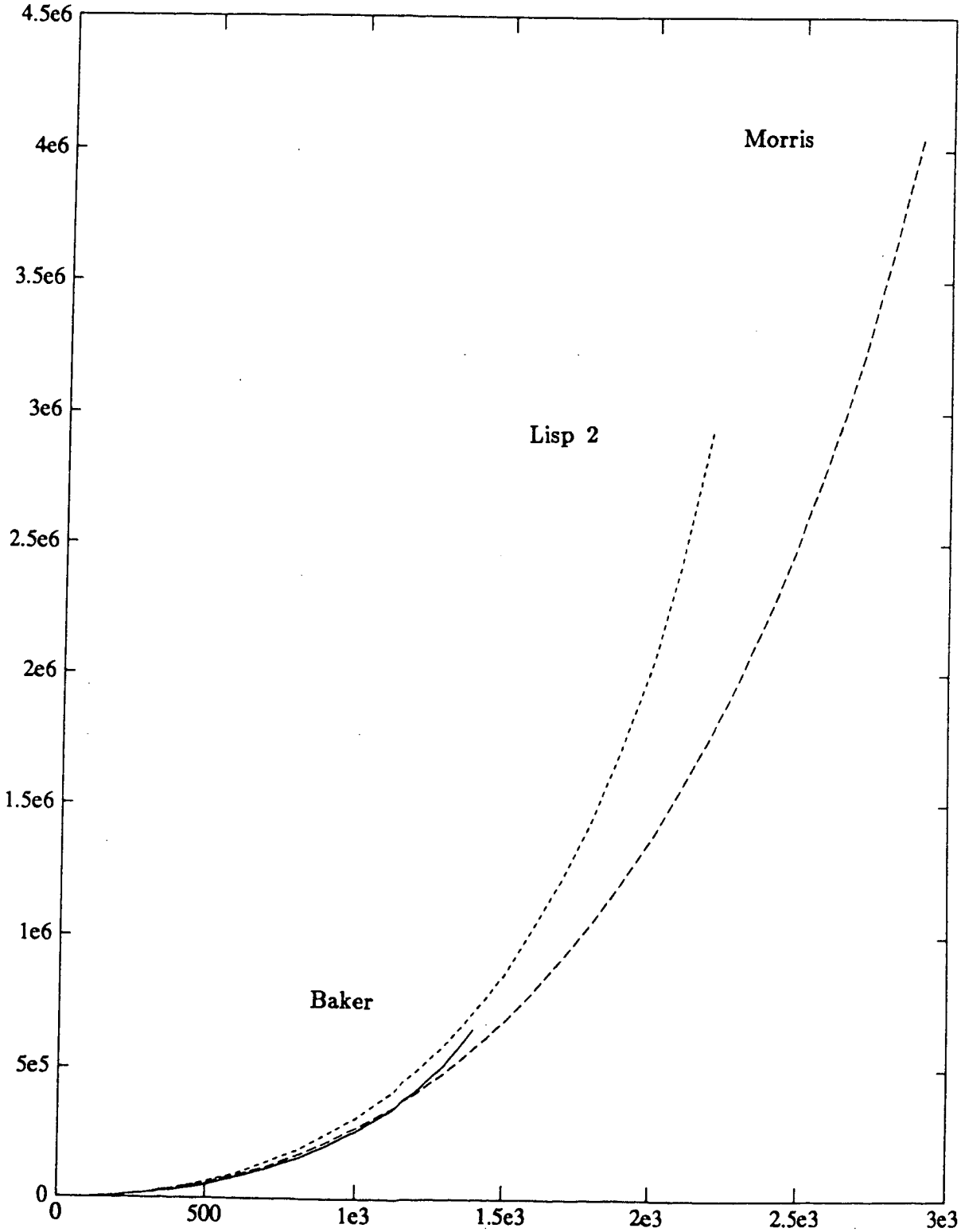


Figure 6 : inversion naïve d'une liste avec 1Mo de mémoire
versions Baker , Lisp 2 et Morris

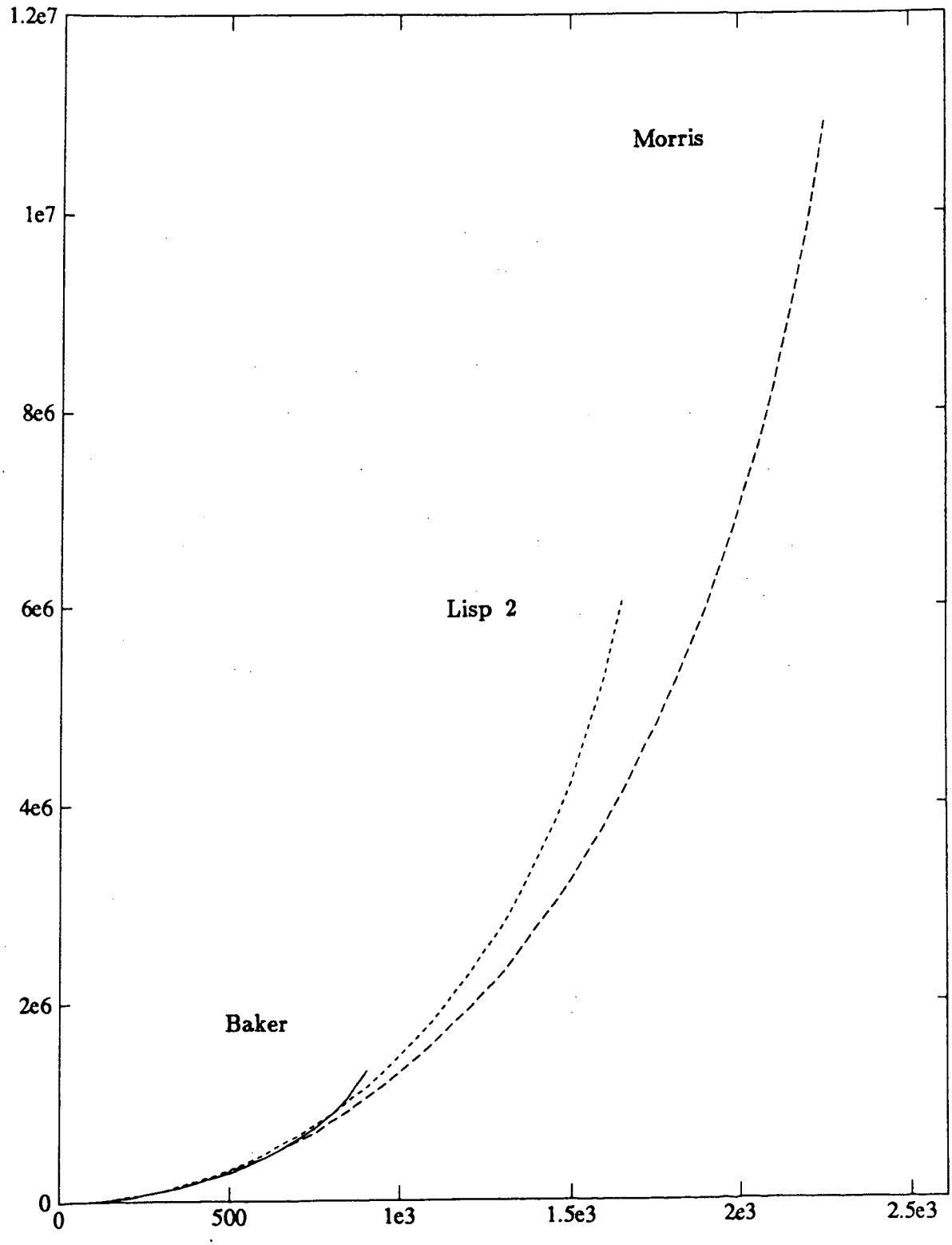


Figure 7 : calcul des nombres premiers avec 1Mo de mémoire
versions Baker , Lisp 2 et Morris

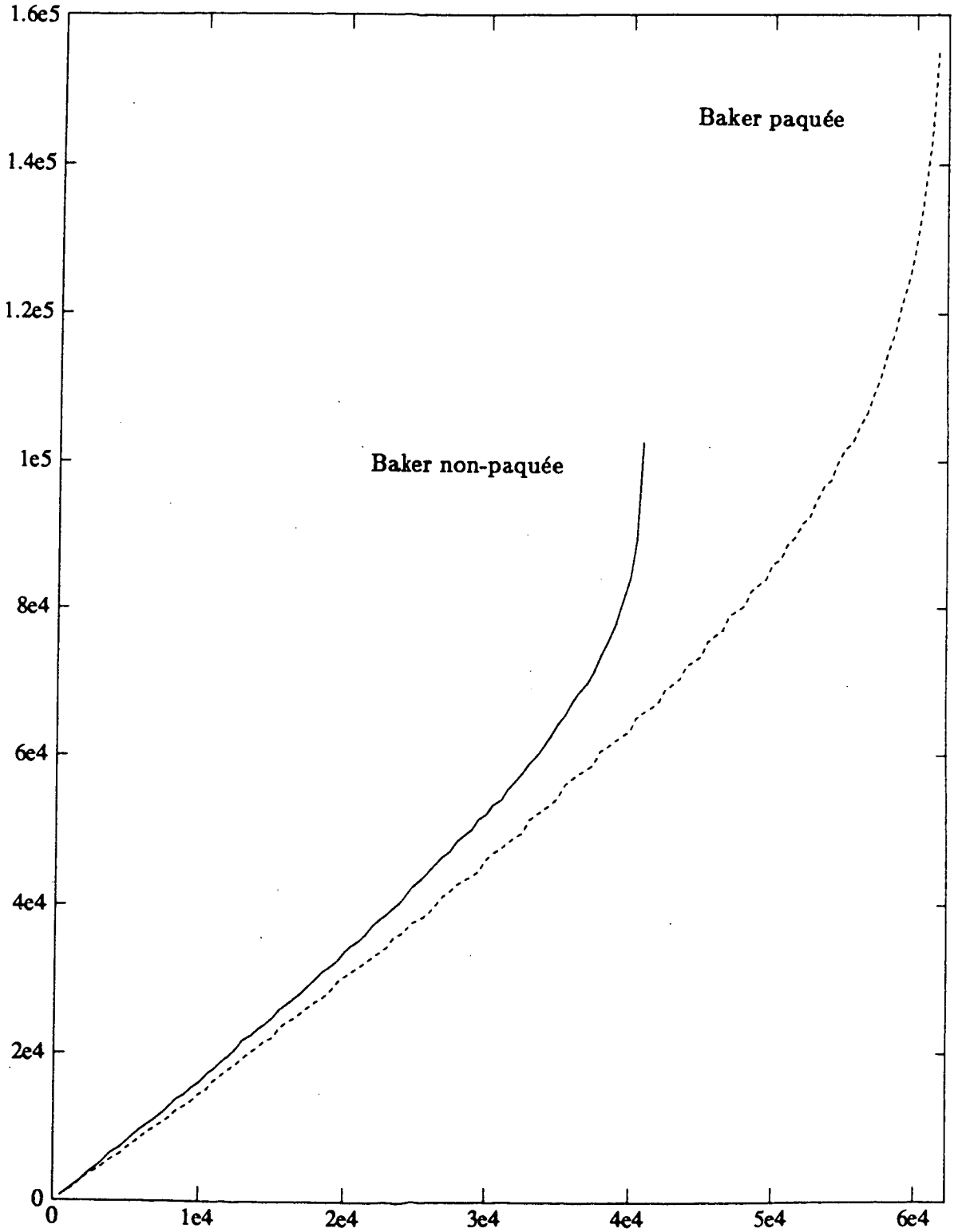


Figure 8 : construction d'une liste avec 1Mo de mémoire
versions Baker et Baker paquée

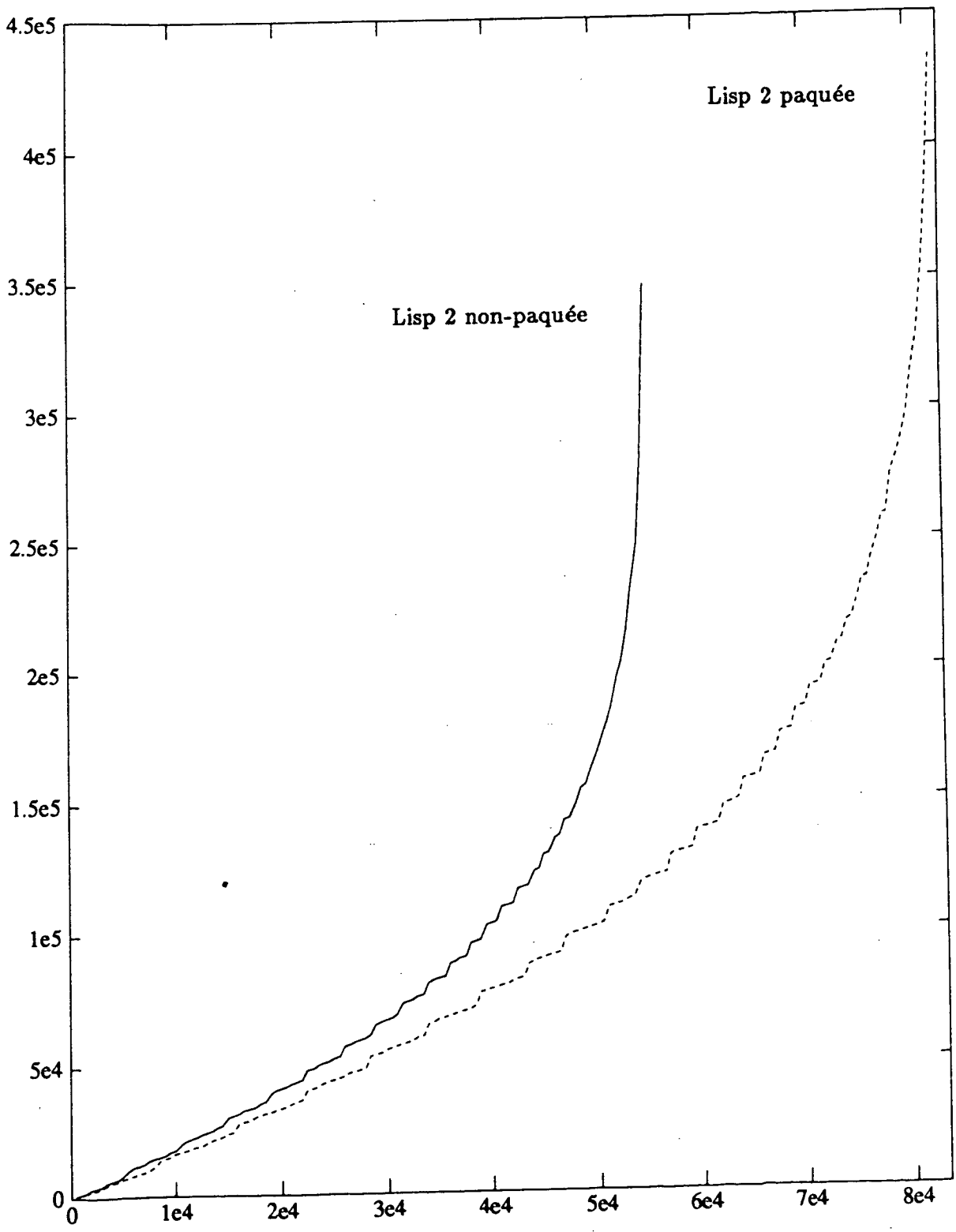


Figure 9 : construction d'une liste avec 1Mo de mémoire
versions Lisp 2 et Lisp 2 paquée

3.3 Performances vitesses brutes

Nous venons d'étudier les limites, de saturation, des différentes versions en leur allouant à tous un espace de même taille. Maintenant, il serait intéressant d'évaluer les performances vitesses de chaque version en équilibrant leurs performances mémoires.

Pour cela, nous allons allouer à chaque version de MALI la taille mémoire nécessaire à la résolution d'un problème de taille donnée.

3.3.1 Calcul de la mémoire à allouer pour un problème de taille donnée

Construction d'une liste de 100000 éléments

Nous voulons que chaque version dispose de l'espace mémoire juste suffisant pour résoudre ce probléme. D'après les calculs effectués dans la section 3.2.1, nous pouvons calculer l'espace à allouer à chaque version.

Le tableau ci-dessous présente ces résultats :

	baker	baker paquée	lisp 2	lisp 2 paquée	morris
taille *Construit* (en octets)	12	8	18	12	12
taille max liste (en éléments)	100000	100000	100000	100000	100000
taille espace alloué (en octets)	2400000	1600000	1800000	1200000	1200000

Nous exécutons maintenant le programme de construction d'une liste en allouant à chaque version de MALI l'espace mémoire calculé.

3.3.2 Comparaison des vitesses d'exécution

Les versions non-paquées

Les temps d'exécution des versions non-paquées Baker , Lisp 2 et Morris en fonction de la taille de la liste construite sont représentés sur la Figure 10.

La première remarque est de constater que les trois versions atteignent la construction d'une liste de 99000 éléments.

Au sujet des vitesses d'exécution, nous constatons que :

la version de MALI avec le récupérateur Baker est la plus rapide de trois versions ;

les versions de MALI avec les récupérateurs Lisp 2 et Morris ont sensiblement les mêmes vitesses d'exécution pour ce programme.

Il est intéressant de noter les différences de comportement lors de l'approche de la saturation de la mémoire. L'utilisation des récupérateurs Lisp 2 et Morris entraîne

un affaiblissement important des performances vitesses lors de l'approche de la saturation. Les récupérations se multiplient avant de détecter la saturation. Le récupérateur Baker aboutit plus rapidement à la détection de cette saturation. Cette différence de comportement provient du choix du seuil de saturation.

les versions paquées

Nous comparons ici les versions paquées et celles non-paquées pour les récupérateurs Baker et Lisp 2 .
Sur la Figure 11 sont rassemblées les deux versions pour le récupérateur Baker .
Sur la Figure 12 sont rassemblées les deux versions pour le récupérateur Lisp 2 .

Nous constatons que pour le récupérateur Lisp 2 , la version paquée est toujours plus rapide que la version non-paquée.

Avec le récupérateur Baker , un phénomène différent se produit : la version paquée est plus rapide au début, mais près de la saturation, elle se retrouve plus lente que la version non paquée. Cela semble indiquer que les performances , par rapport à la taille des objets utiles, de la version paquée se dégradent plus vite que celles de la version non-paquée ; de plus que l'interpréteur est plus rapide avec la version paquée.

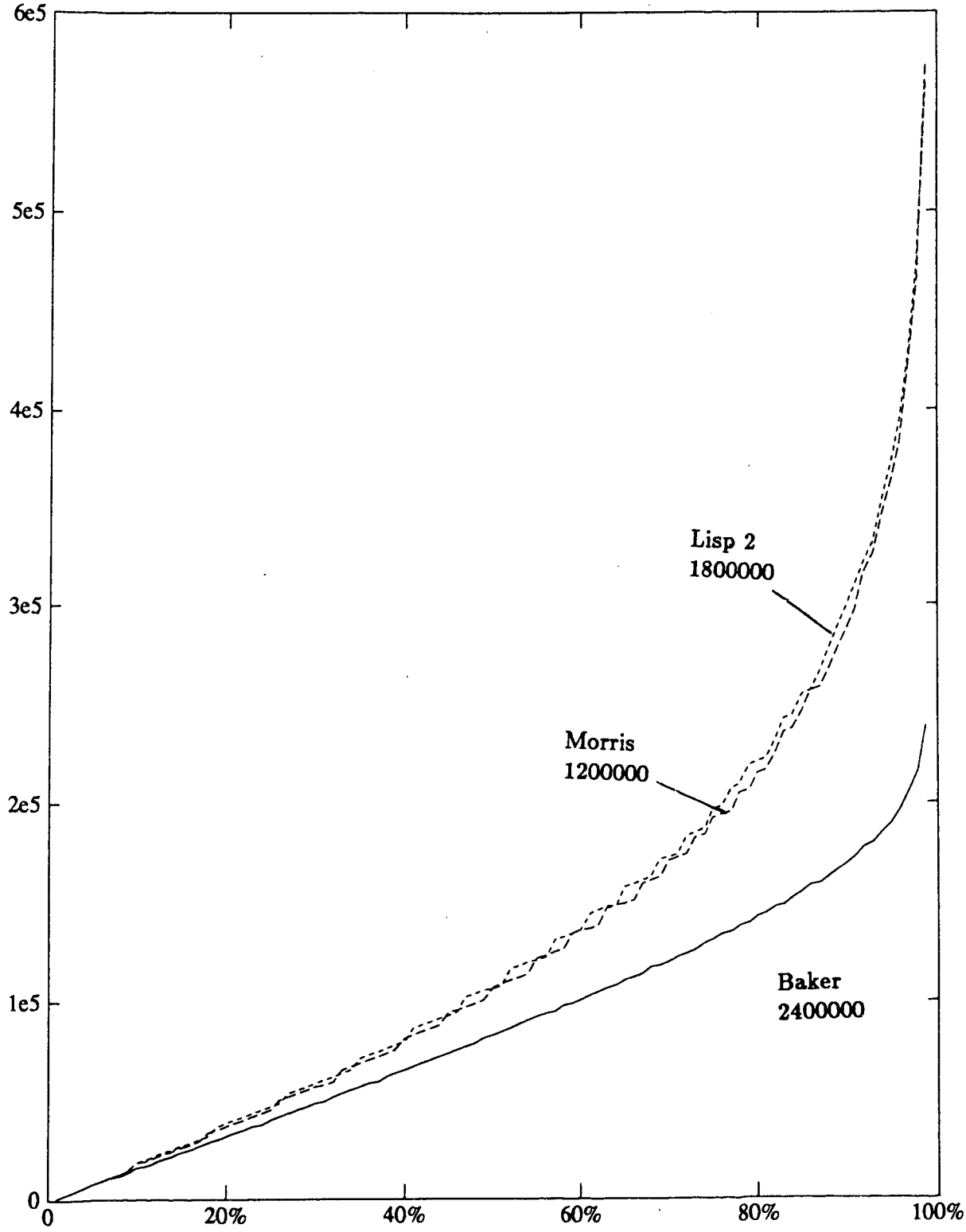


Figure 10 : construction d'une liste avec saturation à 100000 éléments versions Baker , Lisp 2 et Morris

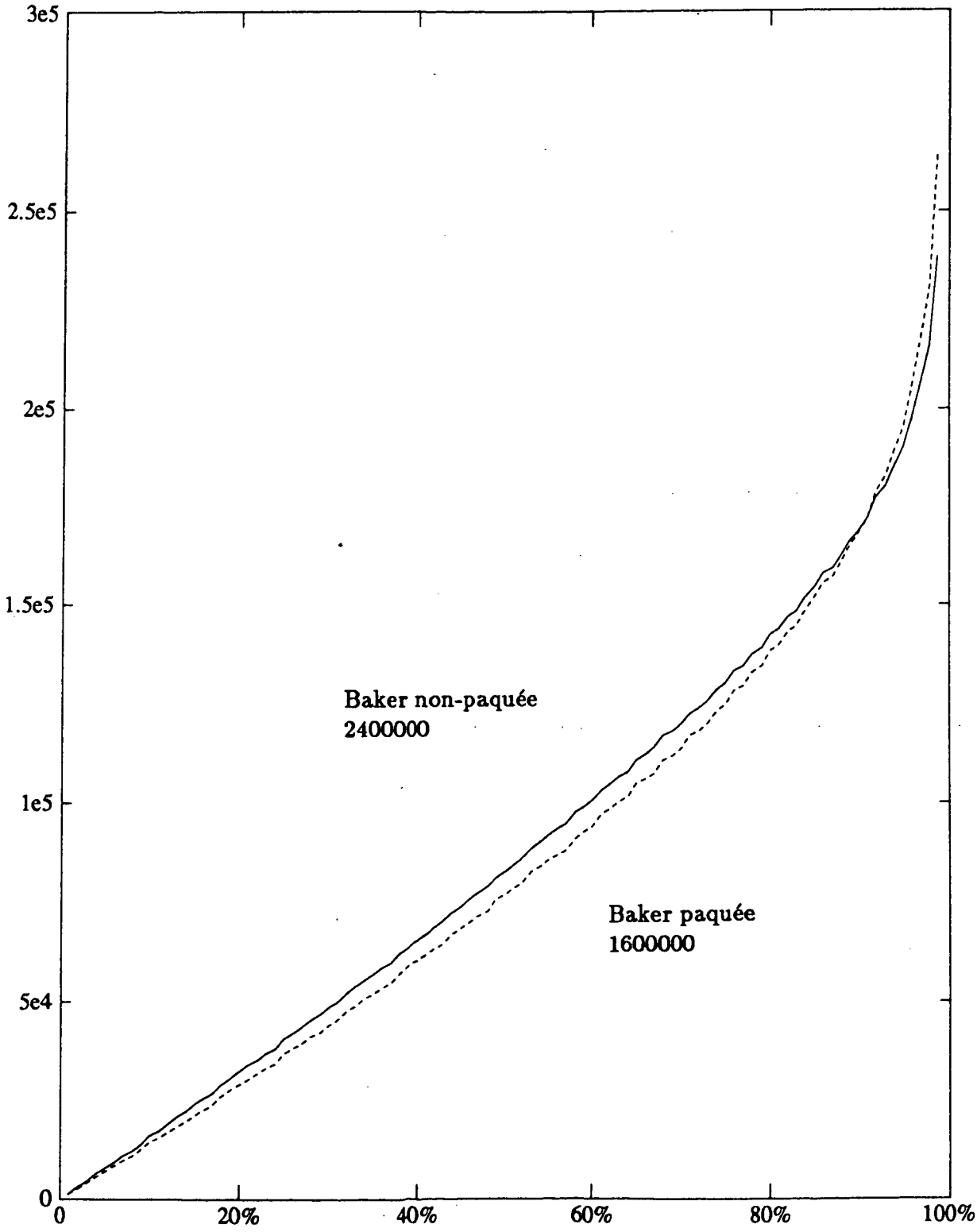


Figure 11 : construction d'une liste avec saturation à 100000 éléments versions Baker et Baker paquée

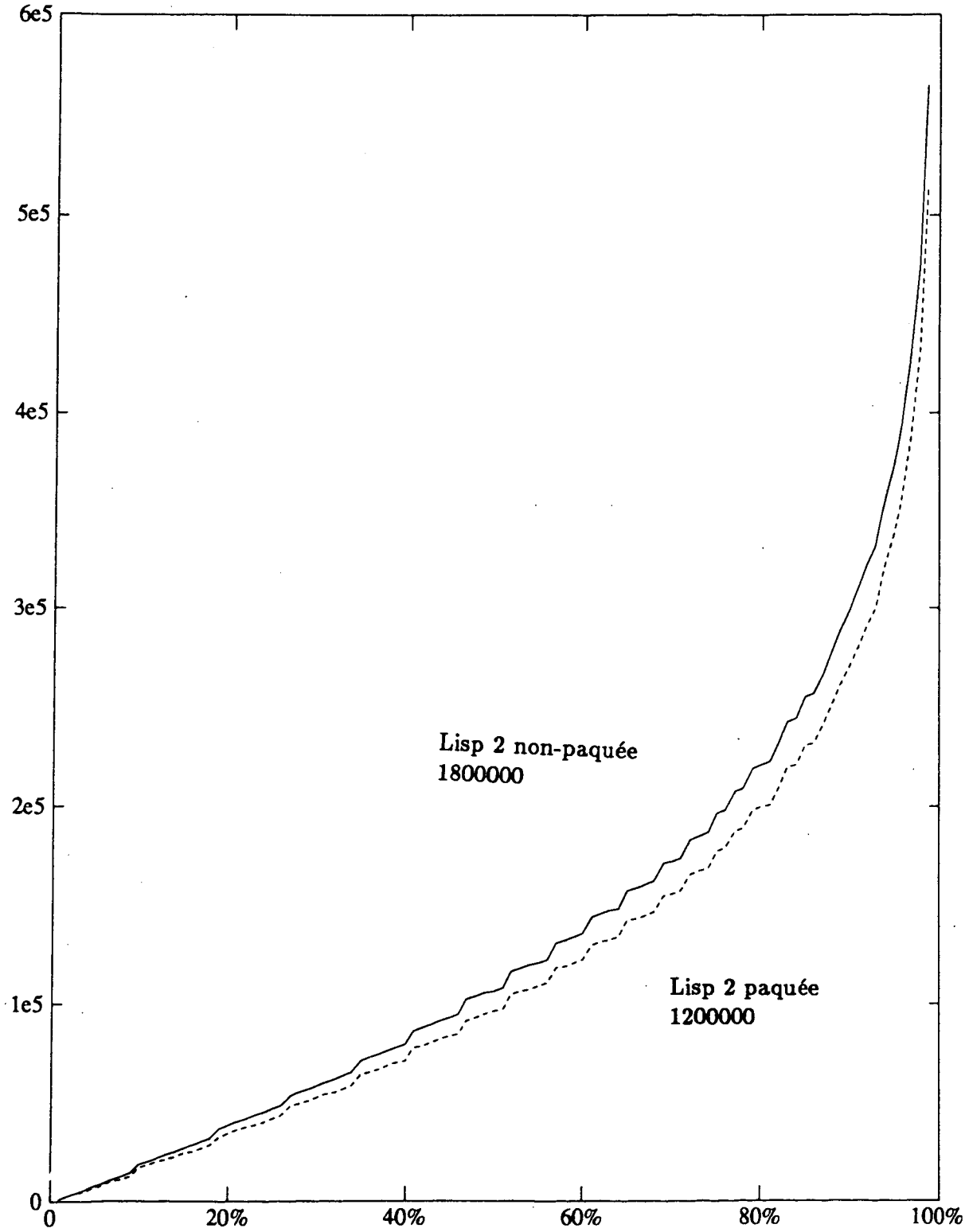


Figure 12 : construction d'une liste avec saturation à 100000 éléments versions Lisp 2 et Lisp 2 paquée

3.4 Analyse détaillée sur le programme de construction d'une liste

Le problème de la création d'une liste va nous permettre de déterminer le temps de récupération en fonction de la taille de la liste construite.

3.4.1 Mesures lors de la construction d'une liste

Nous allouons d'abord à chaque version de MALI un espace mémoire juste suffisant pour construire une liste de 100000 éléments. Puis lors de l'exécution du programme, nous prenons, tous les 500 éléments ajoutés à la liste, les mesures suivantes :

le nombre de récupérations effectuées pour ajouter les 500 derniers éléments ;
le temps nécessaire à ajouter ces 500 éléments ;
le temps écoulé depuis le début de la construction de la liste.

Nous disposons donc d'un tableau de type suivant :

taille de la liste déjà construite	nombre de récupérations	temps partiel	temps total cumulé
500	0	800	800
1000	0	800	1600
1500	1	940	2540
.	.	.	.
.	.	.	.
.	.	.	.

3.4.2 Méthode pour calculer le temps de récupération/taille de la liste

A partir du tableau de mesures précédent nous obtenons deux courbes :

1. le temps nécessaire pour ajouter 500 éléments lorsqu'il n'y a pas de récupération ;
2. le temps nécessaire pour ajouter 500 éléments lorsqu'il y a une récupération.

Le temps de création de 500 éléments supplémentaires dans la liste, sans récupération, est pour chaque version constant par rapport à la taille de la liste.

Le temps de création de 500 éléments supplémentaires dans la liste, avec une récupération, est linéaire par rapport à la taille de la liste pour toutes les versions.

La Figure 13 montre pour MALI avec le récupérateur Baker les temps partiels pour la création de 500 éléments supplémentaires, la courbe du temps de création sans récupération et la courbe du temps de création avec une récupération.

A partir de cela, si l'on fait la différence entre les deux fonctions donnant le temps de création de la liste avec et sans récupération, on obtient le temps de récupération en fonction de la taille de la liste.

3.4.3 Les résultats pour la construction d'une liste de valeurs

La méthode précédente a été appliquée à toutes les versions de MALI .

Les versions non-paquées

Pour construire la liste de 100000 éléments, la version Baker dispose de 2400000 octets, celle de Lisp 2 1800000 et celle de Morris 1200000 .

La Figure 14 représente les fonctions temps de récupération/taille de la liste pour les récupérateurs Baker , Lisp 2 et Morris .

Sur cet exemple :

Le récupérateur Baker apparaît de loin le plus rapide des trois.

Le récupérateur Lisp 2 est légèrement plus rapide que Morris .

Les effets des versions paquées

Il est intéressant de comparer les versions paquées et celles non-paquées.

Cela est fait sur la Figure 15 pour le récupérateur Baker et sur la Figure 16 pour le récupérateur Lisp 2 .

Nous y constatons que la récupération Lisp 2 paquée est toujours plus rapide que la récupération non-paquée. De plus la pente de la courbe pour Lisp 2 paquée est moins importante que celle pour Lisp 2 non-paquée. Ceci explique le fait que lors de la construction de la liste la version paquée reste toujours plus rapide que la version non-paquée (voir Figure 11).

Pour le récupérateur Baker , la récupération paquée est plus lente que celle non-paqué. La pente de sa courbe ést aussi plus importante. Le rapport entre la différence des temps de récupération et la taille de la liste est égal à 0,9%. Cela explique pourquoi lors de l'exécution du programme de construction de liste la version Baker paquée soit d'abord plus rapide que la version non-paquée, puis devienne plus lente en approchant de la saturation (voir Figure 12).

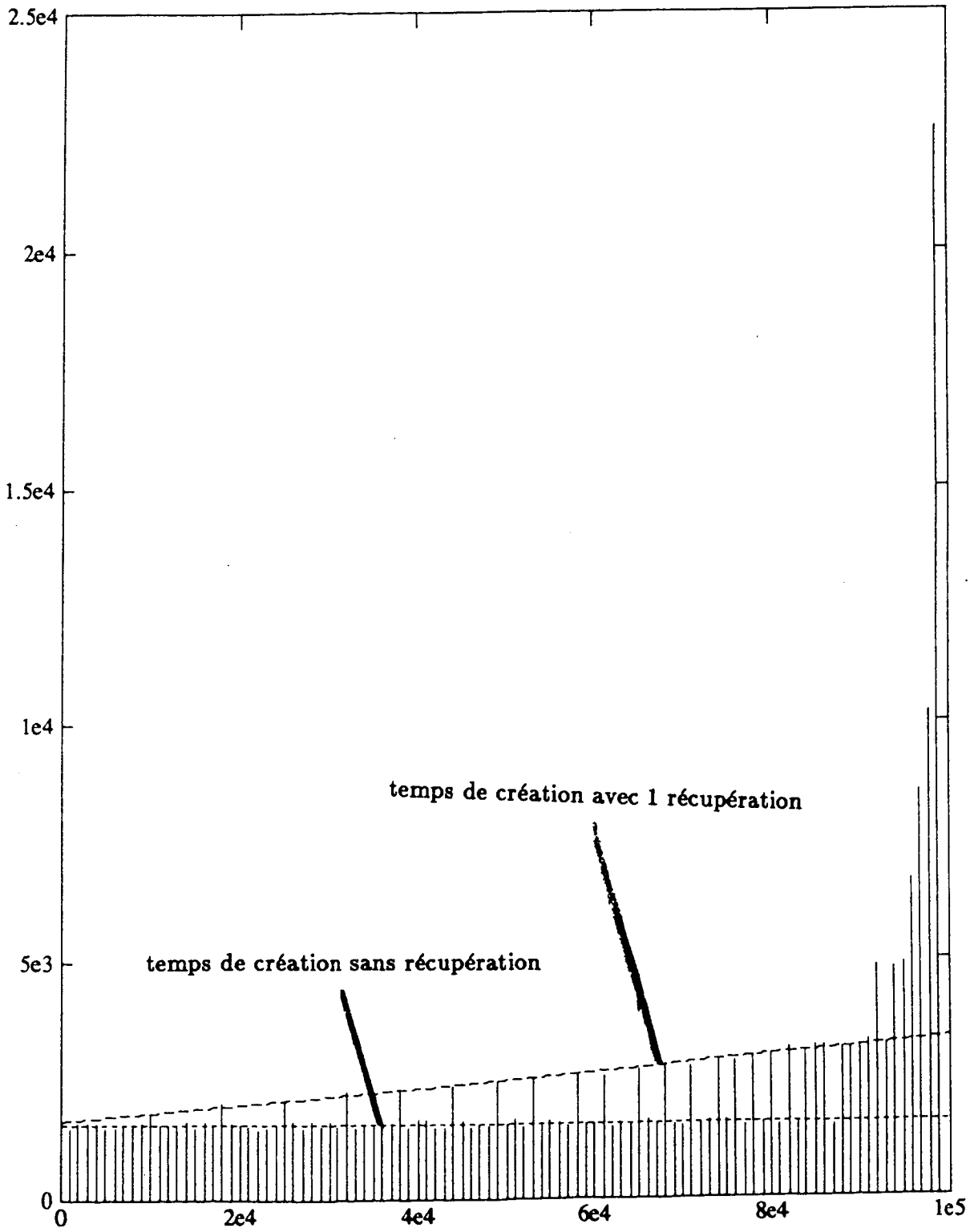


Figure 13 : construction d'une liste avec la version Baker
temps d'exécution pour chaque pas de 500 éléments

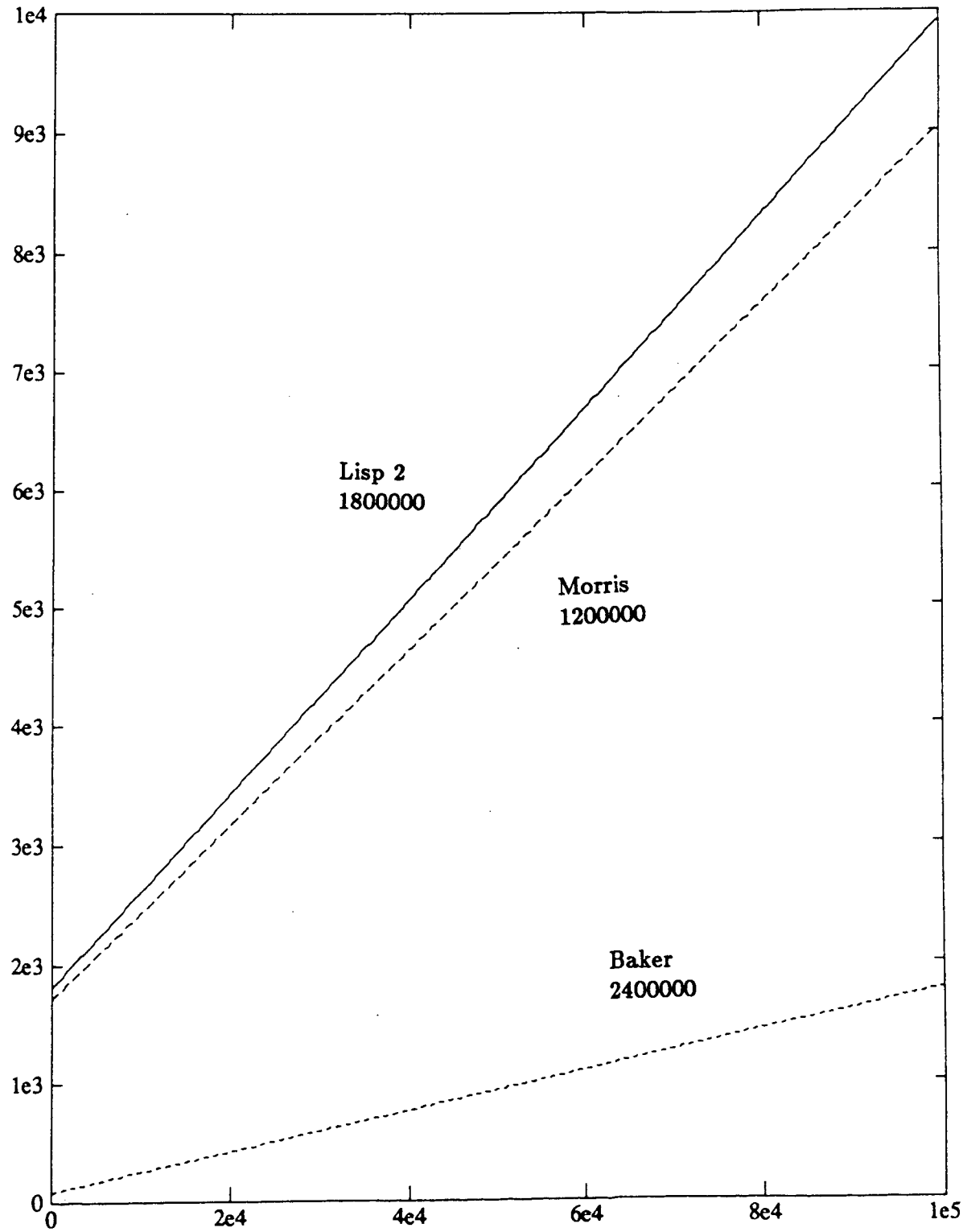


Figure 14 : temps de récupération/taille de la liste
versions Baker , Lisp 2 et Morris

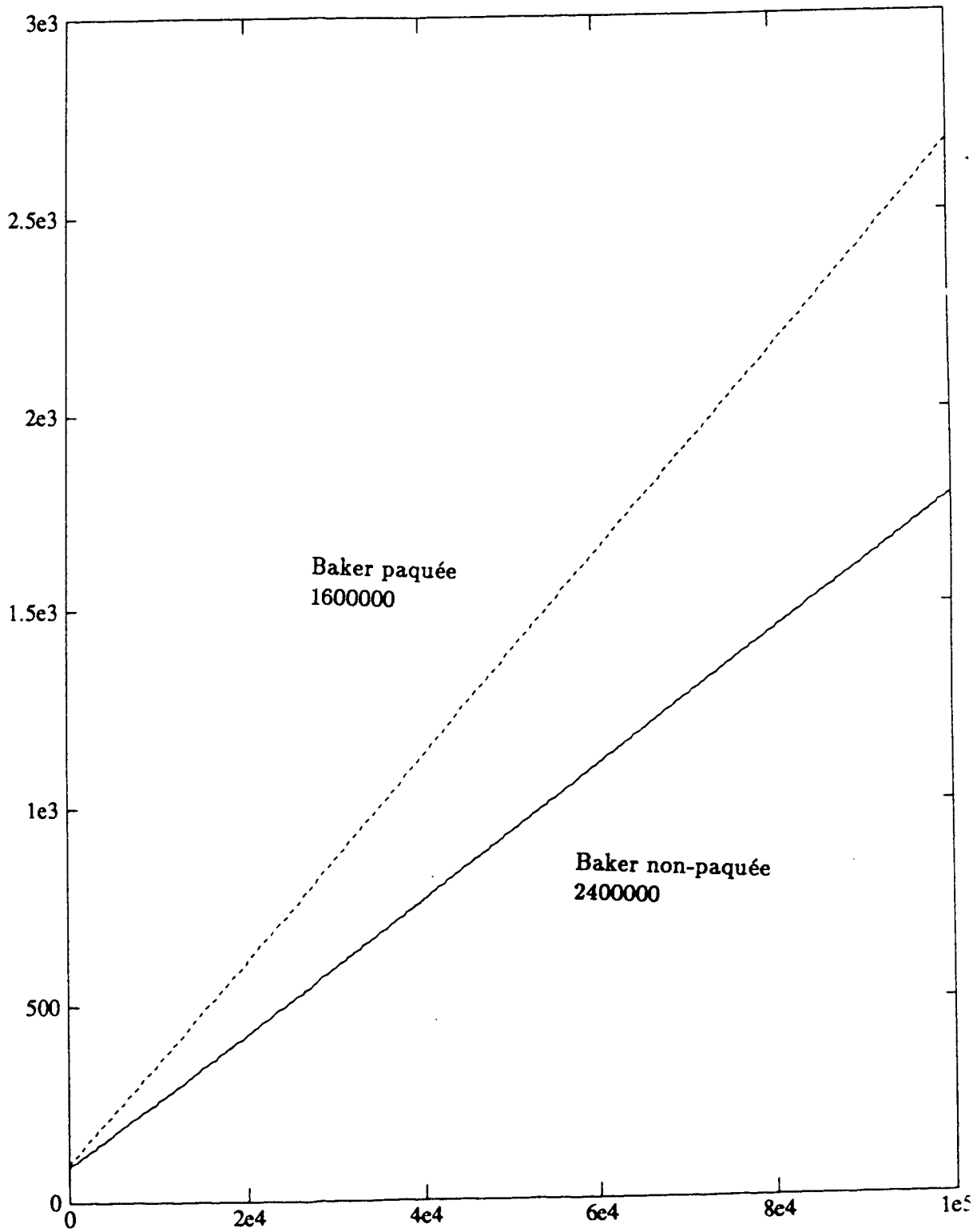


Figure 15 : temps de récupération/taille de la liste
versions Baker et Baker paquée

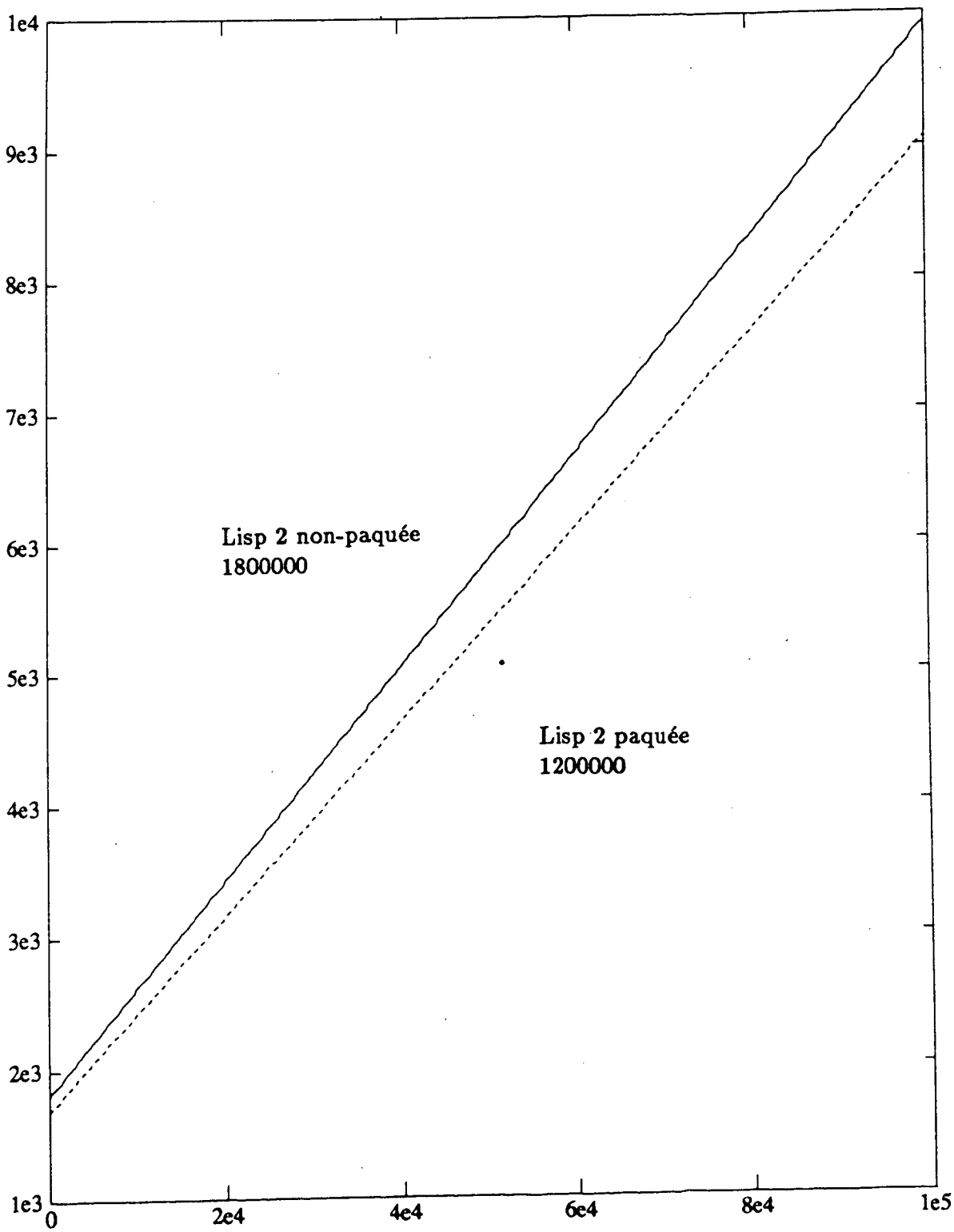


Figure 16 : temps de récupération/taille de la liste
versions Lisp 2 et Lisp 2 paquée

CONCLUSION

La version Morris par rapport à la version Lisp 2

La vitesse pure de la version Morris est supérieure a celle de la version Lisp 2 ; elles est de plus équivalente à la vitesse de la version Lisp 2 paquée.

L'interprétion est ralentie par la version Lisp 2 : surcoût de temps pour la création des termes et pour l'accés aux termes (à cause de la cellule supplémentaire en début d'objet).

La représentation des informations est plus compacte dans la version Morris . A mémoire égale, la taille des problèmes résolus par Morris est supérieure à celle des problèmes résolus par Lisp 2 .

D'aprés ces trois considérations, il apparait donc que la version Morris est toujours plus intéressante que la version Lisp 2 .

La version Morris par rapport à la version Baker

Pour un taille de mémoire égale, la version Morris a, en général, des performances mémoire deux fois supérieures à la version Baker .

Par contre, à taux de saturation égal (saturation pour un problème de taille donné), la version Baker est nettement plus rapide que celle Morris .

Le problème de récupération mémoire au retour-arrière (par libération de la place des objets créés depuis le dernier point de reprise) ne handicape pas la version Baker . Elle ne peut pas toujours effectuer cette récupération mais cela ne semble pas arriver souvent.

Le confort d'utilisation est meilleur avec Baker :

- les à-coups, lors de l'interprétation, provoqués les récupérations mémoire sont moins sensibles (la complexité de Baker est proportionnelle à la taille des objets utiles tandis que Morris effectue des parcours linéaires de la mémoire) ;

- la saturation est plus rapidement détectée. La valeur du seuil de saturation apparait mieux adaptée à Baker qu'à Morris . Un abaissement de ce seuil dans la version Morris permettrait une détection plus rapide de la saturation, mais risquerait en contre partie de diminuer nettement les performances mémoire.

Le choix entre Morris et Baker se révèle donc difficile. Il pourrait dépendre de la taille de l'espace dont peut disposer MALI . En effet, il semble préférable d'utiliser Baker (pour ses performances vitesse) lorsque la mémoire de MALI peut être grande et d'utiliser Morris (pour ses performances mémoire) lorsque la mémoire est limitée.

Références

[1] H.G. BAKER : "List-processing in real-time on a serial computer". *Commun.* ACM 21, 4 (Avril 1978).

[2] Y. BEKKERS, B. CANET, O. RIDOUX, L. UNGARO : "MALI : A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages". Proc. of the 2nd Int. Logic programming, IEEE, Sept. 1986, Salt-Lake City, USA.

[3] KNUTH, D.E. "The Art of Computer Programming", vol. 1 : " Fundamental Algorithms". Addison Wesley, Reading, Mass., 1973.

[4] MORRIS, F.L. "A time and space efficient garbage compaction algorithm" . *Commun.* ACM 21, 8 (Aout 1978), 662–665.

[5] MORRIS, F.L. "On a comparison of garbage collection techniques". *Commun.* ACM 22, 10 (Octobre 1979), 571.

[6] O. RIDOUX, "Gestion de mémoire temps-réel des langages de programmation relationnelle." Thèse, Université de Rennes I, 1986.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 438 **A PROPOS DE LA RESOLUTION D'UN SYSTEME LINEAIRE DANS UN CORPS FINI : ALGORITHMES ET MACHINES PARALLELES**
Hervé LE VERGE, Patrice QUINTON, Yves ROBERT, Gilles VILLARD
22 Pages, Novembre 1988.

PI 439 **ALPHA DU CENTAUR : A PROTOTYPE ENVIRONMENT FOR THE DESIGN OF PARALLEL REGULAR ALGORITHMS**
Pierrick GACHET, Patrice QUINTON, Christophe MAURAS
Yannick SAOUTER
20 Pages, Novembre 1988.

PI 440 **CONSTRUCTION METHODIQUE D'UN ALGORITHME REPARTI DE DETECTION DE LA TERMINAISON**
Jean-Michel HELARY, Michel RAYNAL
18 Pages, Décembre 1988.

PI 441 **LES GRAPHES A MOTIFS**
Didier CAUCAL
46 Pages, Décembre 1988.

PI 442 **CAUSAL TREES**
Philippe DARONDEAU, Pierpaolo DEGANO
44 Pages, Décembre 1988.

PI 443 **TROIS IMPLANTATIONS DU RECUPERATEUR DE MEMOIRE DE LA MACHINE MALI**
Michel LE HENAFF, Hervé SANSON
118 Pages, Décembre 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique